COBB

January 1997 • $11.50
VOL. 3 NO. 1

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris™

in this issue

1
Setting up virtual Netscape Web servers

4
Packaging groups of files for distribution

8
Execute large jobs during the quiet times on your system

10
An introduction to Perl

14
Using associative arrays in Perl

16
Summertime... in the dead of winter

16
Revisiting the xterm title bar

Setting up virtual Netscape Web servers

By Jerry L. M. Phillips, M.S.

Virtual-serving is a technique that allows multiple virtual servers to function as separate entities on a single UNIX server. Each virtual server can provide one or more virtual services, such as WWW and ftp, that are specific to that virtual server. Thus, one machine can appear to operate as several different machines running entirely different server applications. In this article, we'll describe the implementation of virtual World Wide Web servers on a Solaris 2.4 platform, using Netscape Communications Server, v1.1.

DNS configuration

Netscape distinguishes between a physical server that corresponds to the physical interface on your UNIX server and virtual servers that correspond to virtual interfaces on your UNIX server. We'll use that naming convention throughout this article.

In our example, we'll configure two Web servers on one UNIX server. For the first step, make an entry in your domain name server (DNS) for your physical server, followed by an entry for your virtual server. For our example, we've entered *www.physical.com* as the fully qualified domain name and *157.21.201.1* as the IP address of the physical server. Then, we entered *www.virtual.com* and *157.21.201.2* for the virtual server. Please note that you can add multiple virtual servers, but in this example, we're only adding one. Next, confirm that `nslookup` can detect both entries before proceeding. (Note that while we're using two different domain names, both IP addresses are from the same class B license.)

Virtual interface configuration

Normally, when you install Solaris, you tell it the network address that it responds to. For the purposes of our example, we originally configured the machine whose name is *physical* and told it that it's part of the *.com* domain, with an IP address of *157.21.201.1*.

Just to verify our network setup, let's use the `ifconfig` command to display the network configuration of *physical.com*. To do so, just enter the `ifconfig -a` command, as shown in Figure A on page 2.

As you can see, physical currently has two network interface drivers. The first, lo0, is the loopback address that doesn't correspond to any hardware. The second, le0, represents the Ethernet card. You can see its IP address (157.21.201.1) and Ethernet address (8:0:20:59:a4:6c).

Now here's the tricky part: We're going to tell the le0 driver to answer to another IP address,

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris

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

Prices
U.S. ..... $115/yr ($11.50 each)
Outside U.S. ..... $135/yr ($16.95 each)

Phone and Fax
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) 491-4200
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 group subscriptions, address your letters to:

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

Staff
Editor-in-Chief ..... Marco C. Mason
Contributing Editors ..... Al Alexander
Jerry L. M. Phillips
Production Artists ..... Margueriete Winburn
Liz Palmer
Editor ..... Karen S. Shields
Publications Coordinator ..... Linda Recktenwald
Marketing Coordinator ..... Marcella Able
Circulation Manager ..... Mike Schroeder
Editorial Director ..... Linda Baughman
VP/Publisher ..... Lou Armstrong
President ..... John A. Jenkins

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, or American Express, or we can bill you.

Postmaster
Periodicals postage paid in Louisville, KY.
Postmaster: Send address changes to:

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

Figure A

```
#
lo0:flags=849<UP,LOOPBACK,RUNNING, MULTICAST> mtu 8232
      inet 127.0.0.1 netmask ff000000
le0:flags=863<UP,BROADCAST, NOTRAILERS,RUNNING, MULTICAST> mtu 1500
     inet 157.21.201.1 netmask ffffff00 broadcast 157.21.201.255
     ether 8:0:20:59:a4:6c
```

The `ifconfig -a` command shows the status of all your network interfaces.

while keeping its old one. This way, you can communicate with the computer at two different addresses! This is the heart of the virtual-server technique.

To configure the second IP address, which will represent your virtual server, use the following command:

```
# ifconfig le0:1 157.21.201.2
  ➥netmask 255.255.255.0 up
```

This tells the `ifconfig` command to add a new logical IP address to the same card controlled by le0 and calls it le0:1. Once you've set up your new IP address, you can verify your entry by again entering the `ifconfig -a` command. Solaris can accommodate up to 255 logical units per network interface, e.g., le0:1, le0:2, le0:3, etc. Conceivably, you could set up 255 virtual servers on one network interface card with 255 different IP addresses.

Before proceeding, confirm that you can `ping` the physical and virtual servers and that the IP addresses match the ones you put in the DNS. Be sure to `ping` the physical and virtual servers from another machine: `ping`ing the virtual server from the machine that hosts it doesn't prove anything.

Please note that you'll need to start your logical interface each time you start Solaris. To do so, you'll want to put this command in a startup script. We'll discuss this in the Cleanup section.

Netscape installation

Here, we assume that you have some familiarity with the installation of Netscape server products. Unfortunately, if you've already installed Netscape Communications Server as a physical server-only configuration, you'll have to reinstall it.

Physical-server configuration

Begin by installing the physical server. Don't install the virtual server first! You must install the administrative server during the first server installation for this technique to succeed. Issue the command `ns-setup`. Type *www.physical.com* in response to the Full Name: prompt and specify the location of your Netscape client in response to the Network Navigator: prompt.

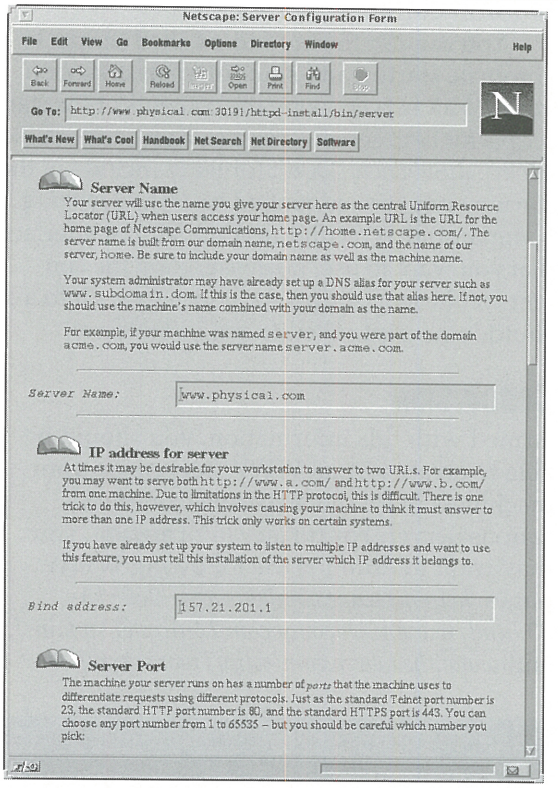
The Server Configuration Form contains three critical entries. Enter *www.physical.com* as the Server Name and *157.21.201.1* as the Bind Address, as shown in Figure B.

Set the Server Location using */export/home/www/ns-home*, as shown in Figure C. Server Location represents the installation point for the Netscape Communications Server software.

There is one critical entry in the Document Configuration Form. Enter */export/home/www/ns-home/physical* as the Document Root for the physical server, as shown in Figure D. This is your directory for all documents relating to the physical server. When we set up virtual servers, each will have its own Document Root directory, so the default Web page for each server will be different.

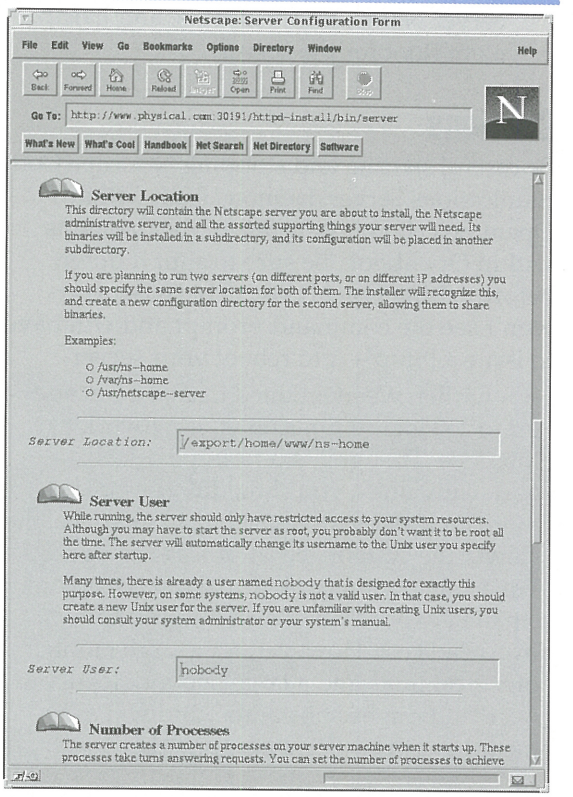
Complete the Administrative Configuration Form at this time to suit your taste. Confirm that your physical server and administrative server are running. Then shut down both servers.

Figure B



This part of the Server Configuration Form allows you to set the server name and bind address for the server.

Figure C



The Server Location specifies the location of all the server support files.

Virtual-server configuration

Restart `ns-setup` to install your virtual server. Type *www.virtual.com* in response to the Full Name: prompt and specify the location of your Netscape client in response to the Network Navigator: prompt.

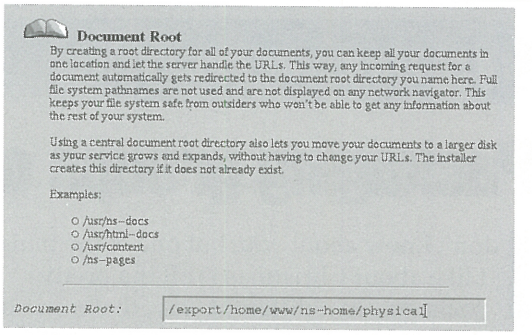
For this server, we're going to use our virtual server name and IP address. In the Server Configuration Form, enter *www.virtual.com* as the Server Name and *157.21.201.2* as the Bind Address. Go ahead and set the Server Location again using */export/home/www/ns-home*, just as before.

Now we must provide an alternate Document Root for the new virtual server. To do so, use */export/home/www/ns-home/virtual* as the Document Root for your virtual server. This is your directory for all documents relating to the virtual server.

Don't redo the Administrative Configuration Form following installation of a virtual server. That step is necessary only during installation of the physical server.

Finally, confirm that your virtual server runs correctly. Restart your physical server and confirm that both servers can access each other. Then restart the administrative server and confirm that it sees both the physical and virtual servers.

Figure D



The Document Root specifies the home directory on your WWW server.

Adding more virtual servers

You can create additional virtual servers by following these steps:

1. Add an appropriate entry to your DNS for the new virtual server.
2. Configure a new virtual interface on your network interface.

3. Configure the virtual server using `ns-setup`. Remember to specify a different Server Name, Bind Address, and Document Root for the server.

Cautions

Be careful starting and stopping Netscape Communications Server daemons from within OpenLook. If you start one of the daemons within OpenLook, be sure to shut it down before exiting OpenLook. Otherwise, you won't see the command prompt and will have to issue a [Stop]+A to reboot your server.

Earlier, we mentioned that you can add up to 255 logical units on your network interface. Though it's certainly possible to add many units, you shouldn't do so. The more logical units you add to an interface, the slower the system gets at processing packets. For a small number of virtual interfaces, or a small network load, it's an insignificant overhead cost. However, for a large number of logical units and a large network load, the overhead can be substantial, and you can incur long delays processing packets.

Cleanup

Our example leaves you with two directory structures containing shell scripts and log files for the physical and virtual servers, i.e., */export/home/www/ns-home/httpd-80.157.21.201.1* and */export/home/www/ns-home/httpd-80.157.21.201.2*.

You can use the individual shell scripts to automate startup and shutdown processes on your UNIX server. Copy the contents of the start scripts for both servers into a startup script called */etc/rc2.d/S99www*. Don't forget to include in the startup script the command line (the `ifconfig` command) that configures each virtual interface that you want to use. Likewise, copy the contents of the stop scripts for both servers into a shutdown script called */etc/rc2.d/K99www*.

Conclusion

Armed with this information, you should be able to create virtual WWW servers on your machine(s) with the Netscape Communications Server. For further information, you may want to examine the following WWW pages: *help.netscape.com/kb/server/960513-83.html*, *www.LANcomp.com/MultipleDomains/*, and *www.thesphere.com/~dip/TwoServers/*. ❖

Jerry L. M. Phillips, M.S., is Director of the Academic Computer Center at Eastern Virginia Medical School. Besides his administrative duties, he manages Sun/Solaris-based platforms for the medical school, including DNS, WWW, anonymous ftp, and Usenet News servers.

CREATING PACKAGES

Packaging groups of files for distribution

I don't know about you, but one of the things I like about Christmas is all the pretty packages. In a different way, that's one of the things I like about professionally produced software. I know that it will arrive all dressed up in a nice package. I won't have to struggle to install it. I can simply use the `pkgadd` command to install the software.

If you're like me, you've probably wondered how those packages are put together. You've tried to gain some insight by reading the `man` pages on the `pkgadd` command. Again, if you're like me, you may have even wanted to give up in disgust, looking for a better way to create a package.

While the `man` pages aren't very clear, it gets much easier if you examine a package and watch it being produced. It turns out that if you take apart someone else's package, then read the `man` pages a couple of times, it's possible to create your own package. However, in this article, I'm going to save you all that trouble. I'll show you a step-by-step procedure you can use to create a package. Once you see it, it's easy to make your own packages.

Create a clean directory structure

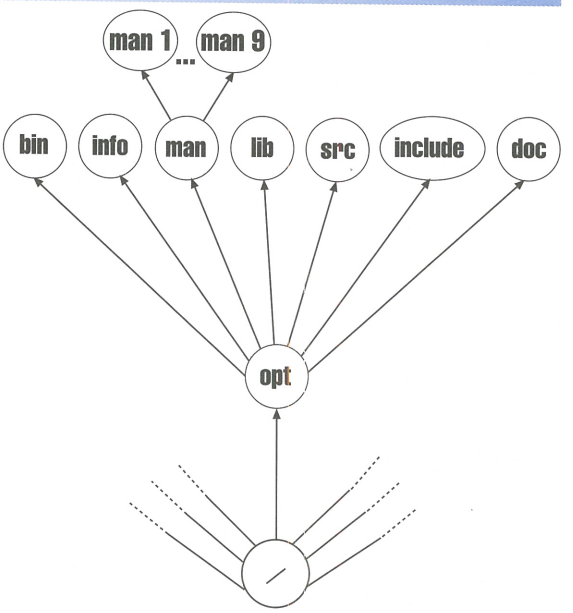
In order to make your packages as easy to use as possible, you should use a standardized directory structure. For example, many packages install themselves into the */opt* directory, placing executable files in the */opt/bin* directory, `man` pages in */opt/man*, etc.

Using a standardized directory structure helps keep the end user's machine clean and

allows the end user to predict where your files are going. It also helps you and your end users when you uninstall the package.

When you install many of the GNU tools, you'll notice that they have a directory hierarchy like the one shown in Figure A. The executables are in the *bin* branch, the `man` pages (which GNU is using less and less) in *man*, the information pages in *info*, the libraries in *lib*, the source code in *src* and *include*, and other documentation in *doc*.

Figure A



Many GNU tools use a directory hierarchy like this one.

When you create your package, you don't want to get your data confused with the data in other packages you've installed or are developing. So, you'll want to create your own directory hierarchy with the files you want to package.

You can do this by explicitly creating a base directory for your package, then creating the branch directories. Next, populate your directories with the files you want to package.

For our example, let's create a base directory named *pkgBase*, with the branches *bin* and *man*. Please note that we create a *man1* branch in *man* so we can place our `man` page in the correct location:

```
# mkdir /pkgBase
# mkdir /pkgBase/bin
# mkdir /pkgBase/man
# mkdir /pkgBase/man1
```

Now, let's populate the directory with the files we wish to package. Since we're just doing an example, our package won't contain anything very interesting. We'll just put the `head` and `tail` programs and their `man` pages in our package:

```
# cp /usr/bin/head /pkgBase/bin
# cp /usr/man/man1/head.1 /pkgBase/man/man1
# cp /usr/bin/tail /pkgBase/bin
# cp /usr/man/man1/tail.1 /pkgBase/man/man1
```

Finally, look at your directory hierarchy to ensure that it contains all the correct directory entries and files:

```
# ls -R /pkgBase
/pkgBase:
bin man

/pkgBase/bin:
head tail

/pkgBase/man:
man1

/pkgBase/man/man1:
head.1 tail.1
```

Create the prototype file

Now that we've created our directory hierarchy and loaded it with the files we want to package, we can start creating our package. First, go to the root of your package's base directory, */pkgBase* in this case:

```
# cd /pkgBase
```

Next, we'll use the `pkgproto` command to generate a *prototype* file for our package. This *prototype* file simply lists the files and directories, along with their desired owner and group IDs and permissions:

```
# find . | pkgproto >prototype
```

The `find` command locates all file and directory names in your base directory and all branches and sends their names to the `pkgproto` command. The `pkgproto` command then reads each entry and transforms it into the syntax required by the `pkgmk` command. The result of running this command is as follows:

```
d none man 0755 root other
d none man/man1 0755 root other
f none man/man1/tail.1 0444 root other
f none man/man1/head.1 0444 root other
d none bin 0755 root other
f none bin/tail 0555 root other
f none bin/head 0555 root other
```

Each line beginning with a `d` describes a directory that's in the package, and each line starting with an `f` describes a file. The third column lists the name of the file or directory. The fourth column tells which permissions the file or directory should have, and the last two columns tell us the user and group ID of the file or directory.

In order to make the package as portable as possible, you don't want to have unknown user and group IDs for your files and directories. You also don't really want to use root and other as shown above. You should modify the user and group IDs to bin and bin respectively. You can create and edit this file with the command

```
# find . | pkgproto | awk '{ $5="bin";
  ➥$6="bin"; print }' >prototype
```

After you create and edit your prototype file, you'll also want to add the following line:

```
i pkginfo=/pkgBase/info
```

This line tells the `pkgmk` command which file contains other information describing the package. Don't worry about this file right now, we'll create it in the next section.

Create the information file

Next, you want to create the package information file. In this file, you describe not the file objects that you're installing, but the environment and external information about the package.

If you read the `man` page for the format of the package information file (`man -s4 pkginfo`), you'll see that you can provide quite a bit of information to the end user of your packages. You can even create new information categories, if you like.

Table A

Name	Description
PKG	Package name: Most Sun packages are named SUNW*x*, where *x* is a shorthand name for the package.
NAME	The long name of the package, which can be more descriptive.
ARCH	The architecture associated with the package.
VERSION	The version of the packaged software.
CATEGORY	Which categories the package belongs in (system, application, etc.).
BASEDIR	The target directory on the destination machine.

We'll use these six package parameters in our demonstration package.

For our purposes, we're going to provide just a little information about our demonstration package. Table A shows the parameters we'll use in our demonstration, along with their descriptions.

For our package, we'll create the file */pkgBase/info*, as shown below:

```
PKG="ISOLtest"
NAME="Test package"
ARCH="Sol 2.5 i386"
VERSION="1.00"
CATEGORY="application"
BASEDIR="/opt/ISOLjan97"
```

Thus, we're naming our package ISOLtest, with a more descriptive name of "Test package". The architecture field doesn't seem to have hard-and-fast standards in place, as we've seen "Solaris 2.5", "Sparc", and "i386" in this field. Thus, we've decided to use "Sol 2.5 i386" to tell the user that it's a Solaris 2.5 file on an x86 platform. Since this is our first version, we'll just set the version to 1.00. Finally, we're saying that the default directory for the package on the destination machine is */opt/ISOLjan97*.

Create the package

Now you've finished all the background work. You've created a directory structure that contains the files you want to package, your *prototype* file that describes all the files you want in your package, and the */pkgBase/info* file that describes the package. You're now ready to create the package.

To create your package, you first run the `pkgmk` command. The `pkgmk` command takes your packaging information and all your files and builds a directory in your */var/spool/pkg* directory named *ISOLtest*. This directory contains the files *pkginfo* and *pkgmap*, as well as the directory *reloc*.

The `pkgmk` command creates the *pkginfo* file from the information found in the */pkgBase/info* file and creates the *pkgmap* file from your *prototype* file. The *reloc* directory contains the directory tree that you've created, but with the file owner and group IDs set to bin and bin, respectively.

You can run the `pkgmk` command like this:

```
# pkgmk -r '/pkgBase' -o
## Building pkgmap from package prototype file.
## Processing pkginfo file.
WARNING: parameter <PSTAMP> set to
"Ringo961030055236"
WARNING: parameter <CLASSES> set to "none"
## Attempting to volumize 7 entries in pkgmap.
```

```
part 1 -- 54 blocks, 11 entries
## Packaging one part.
/var/spool/pkg/ISOLtest/pkgmap
/var/spool/pkg/ISOLtest/pkginfo
/var/spool/pkg/ISOLtest/reloc/bin/head
/var/spool/pkg/ISOLtest/reloc/bin/tail
/var/spool/pkg/ISOLtest/reloc/man/man1/head.1
/var/spool/pkg/ISOLtest/reloc/man/man1/tail.1
## Validating control scripts.
## Packaging complete.
```

Please notice that `pkgmk` issues two warning statements. These are harmless. They tell us that we didn't set the `PSTAMP` or `CLASSES` variables in our *info* file. Since `pkgmk` sets them to a reasonable default, we omitted them. The default value of `PSTAMP` is the hostname of the machine on which you build the package, with the date and time appended to it without punctuation.

The `-r directory` option on `pkgmk` tells it the root directory you're using to build your package. This switch, in combination with the `BASEDIR` value in the *info* file, helps you create a package in one place that will install in another place. Simply replace `directory` with the root directory in which you're building your package.

Now we can use the `pkgtrans` command to take all the information in the *ISOLtest* directory and turn it into a single file, making it simpler to transport between machines. You can do so like this:

```
# pkgtrans -s /var/spool/pkg /tmp/ISOLtest-1.00

The following packages are available:
 1 ISOLtest   Test package
        (Sol2.5i386) 1.00

Select package(s) you wish to process (or 'all'
    ➥to process
all packages). (default: all) [?,??,q]: 1
Transferring <ISOLtest> package instance
```

Here, we're telling `pkgtrans` to transfer a package found in the */var/spool/pkg* directory to the file */tmp/ISOLtest-1.00*. It presents us with a list of packages found in the directory and allows us to choose one or more packages to put into the file. We select 1 (referring to the ISOLtest package) and press [Enter], and it creates the */tmp/ISOLtest-1.00* file for us.

Testing and distribution

Whew! You've finally done it. You now have a package of files. But don't send it out to all your remote sites just yet. You need to test it to be sure that you haven't forgotten a file. To do so, let's run `pkgadd`, shown in Figure B, and see if it builds the correct directory hierarchy and places the files correctly.

Figure B

```
# pkgadd -d /tmp/ISOLtest-1.00

The following packages are available:
 1 ISOLtest   Test package
        (Sol2.5i386) 1.00

Select package(s) you wish to process (or 'all' to
process all packages). (default: all) [?,??,q]: 1

Processing package instance <ISOLtest> from </tmp/ISOLtest-1.00>

Test package
(Sol 2.5 i386) 1.00

The selected base directory </opt/ISOLjan97> must
exist before installation is attempted.

Do you want this directory created now [y,n,?,q] y
Using </opt/ISOLjan97> as the package base directory.
## Processing package information.
## Processing system information.
## Verifying disk space requirements.
## Checking for conflicts with packages already installed.
## Checking for setuid/setgid programs.

Installing Test package as <ISOLtest>

## Installing part 1 of 1.
/opt/ISOLjan97/bin/head
/opt/ISOLjan97/bin/tail
/opt/ISOLjan97/man/man1/head.1
/opt/ISOLjan97/man/man1/tail.1
[ verifying class <none> ]

Installation of <ISOLtest> was successful.
```

You should test your package by installing it before you distribute it to your remote sites.

Now you're ready to examine the resulting directory hierarchy to ensure that all the files you need are placed correctly. If all goes well, test any programs or scripts to be sure they work correctly. Now you're ready to distribute your packages. Don't forget that you may want to `compress` or `gzip` your package before distribution to make it smaller and fit on fewer floppy diskettes.

Oops! I forgot a file

What happens if you forget a file? Don't panic. First, go ahead and put the file in your directory hierarchy. Then re-create your *prototype* file. (You can just hand-edit it, but it's simpler and less error prone to just re-create it.) Keep in mind that once you create your *prototype* file, it contains an entry for your *info* file. (There wasn't one last time because we created the *info* file *after* we created the *prototype* file.)

Therefore, if you don't want your *info* file included in your package, make sure you remove the file entry that specifies the *info* file.

You can skip the creation of the *info* file, since it already exists, and go directly to the `pkgmk` command. Earlier, when we showed you the `pkgmk` command, we used the `-o` option. We did so because it tells `pkgmk` to overwrite the package if it already exists. You didn't really need that option earlier, but you'll need it when you create your package the second time.

Before you can retest your package, you must remove it from your system, or it will emit an error when you try to add it again with the `pkgadd` command. To remove the package, simply type `pkgrm`, select the package you're building, and let the `pkgrm` command remove it, as shown in Figure C. Now you're ready for testing again.

Figure C

```
# pkgrm

The following packages are available:
...
11 ISOLtest    Test package
          (Sol2.5i386) 1.00
...
Select package(s) you wish to process (or 'all' to
process all packages). (default: all) [?,??,q]: 11

The following package is currently installed:
  ISOLtest    Test package
          (Sol2.5i386) 1.00

Do you want to remove this package? y
## Removing installed package instance <ISOLtest>
## Verifying package dependencies.
## Processing package information.
## Removing pathnames in class <none>
...
## Updating system information.

Removal of <ISOLtest> was successful.
```

Before you attempt to reinstall the package, be sure to remove it with `pkgrm`.

Conclusion

Now that we've presented the basics of creating a package, you should build a couple for the experience. You'll find them useful for many situations. You'll be able to use them to put together packages of script files, Web pages, or anything else you want to give to someone else. If you want to do anything fancy, be sure to read the section 1 `man` pages for the `pkgadd`, `pkginfo`, `pkgmk`, `pkgproto`, and `pkgtrans` commands and the section 4 `man` pages on the `pkginfo` and `prototype` commands. ❖

COMPUTING AFTER HOURS

Execute large jobs during the quiet times on your system

Have you ever wanted to start a job that would take several hours and tie up an enormous quantity of CPU and I/O? If so, you probably didn't want to do it during the day, where it would slow everyone down. Instead, you may have waited until the end of the day and used the `nohup` command to execute your job before you logged out. (For a description of `nohup`, see the article "Allow Your Commands to Continue Running After You Log Out" in the December 1996 issue.) However, if people are still using the computer when you leave for the day, you'll still slow *them* down.

Perhaps you used `crontab` to schedule your program to execute during the quiet time on your system. (We demonstrated `crontab` in the article "Scheduling a Job for Periodic Execution" in the October 1996 issue.) However, that's a bit too much work to schedule a single job. You have to get your `crontab` list, edit it to include your new job, then submit it. Then, after your job is over, you must retrieve it again, edit the `crontab` list to remove the job, and resubmit it.

The at command

Luckily, there's a simpler way—the `at` command provides the simplicity and convenience of `nohup`, so you don't have to worry about leaving your terminal logged in overnight. It also gives you the ability to schedule when the job will run, as does `crontab`.

Using the `at` command is easy. You simply tell `at` the time you want the job to run and specify the contents of the job. You can

do so like this:

```
# at 2230
at> sort </LargeFile | uniq >/LargeFile
at> <EOT>
warning: commands will be executed using /sbin/sh
job 846819000.a at Thu Oct 31 22:30:00 1996
```

Using the command line, we told `at` to schedule a job at 10:30pm. Then, `at` prompts us for command lines to execute with the `at>` prompt. When we're finished entering commands, we enter the [Ctrl]D character, and `at` responds by displaying the message `<EOT>`. Then `at` tells us which shell it's going to use to process the command line. Finally, `at` tells us the job number and when it plans to execute it.

The `at` command is very flexible in accepting time formats. Our example used the 24-hour clock notation. If you prefer, you can also tell the `at` command the time using am and pm. Thus, you could enter `at 1030pm` instead. If you use a four-digit time, `at` assumes you're specifying both the hour and the minute. If you use one or two digits, it assumes you're entering the hour. However, you can't enter a three-digit time. A few examples of legal time specifiers are 1000, 5pm, 1221AM, and 11am.

If you specify a time earlier than the current time, `at` won't schedule your job for immediate execution. Instead, it assumes that you mean to execute your command at that time on the next day. So if it's 5:00pm right now, and you tell `at` to start a job at 4:30pm, it won't start the job until tomorrow at 4:30pm.

Specifying the shell to use

Though it's the default, you're not limited to using the Bourne shell for your jobs. If you'd rather, you can use the Korn or C shell. You can specify the shell to use with the `-s`, `-c`, or `-k` options. These specify the Bourne, C, and Korn shells, respectively. So if you want to execute a command at 9:00pm and use the C shell, you can type

```
# at -c 9pm
```

Command entry

If you're not a great typist, or if you have a significant number of commands to execute, you may not like having to type the command list flawlessly at the `at>` prompts.

It turns out that the `at` command provides a facility to allow you to specify the name of a file that contains your command list. Thus, you can use your favorite editor to create your command list and tell `at` to use it with the `-f` *commandlistfile* option. So if you have a list of commands to execute in the file */tmp/cmdlist*, you can execute it at 3:30am with the command

```
# at -f/tmp/cmdlist 0330
```

Notification by mail

Please note that if your job outputs any data on the standard output or standard error streams, the `at` facility will package all the output together and mail it to you. Conversely, if your job doesn't output any data to these streams, the `at` facility won't send you mail. If you'd like to get mail anyway, so you can see that your job executed, you can add the `-m` option to the command line.

As an example, if you enter the following `at` job, `at` will send you a directory listing of the root directory via mail:

```
# at 0500
at> ls -al /
at> <EOT>
```

However, the following command won't send you any mail:

```
# at 0505
at> ls -al / >/dev/null
at> <EOT>
```

If you want `at` to inform you that your job has run, add the `-m` switch, and you'll get mail even if your job doesn't generate any output, as shown here:

```
# at -m 0510
at> ls -al >/dev/null
at> <EOT>
```

Listing and removing jobs

If you'd like to see a list of the jobs you've submitted, you can use the `-l` option on the `at` command. When you do so, you won't get a lot of information about the command other than who submitted it, the job number, and the time it will execute. You'll get no further details. Figure A shows a sample of the typical output you can expect.

Figure A

```
# at -l
user = root     846819000.a     Thu Oct 31 22:30:00 1996
user = root     846820200.a     Thu Oct 31 22:50:00 1996
user = root     846813600.a     Thu Oct 31 21:00:00 1996
user = root     846837000.a     Fri Nov  1 03:30:00 1996
```

The `at -l` command shows only who submitted it, the job ID, and the time it will run—not what the job will do.

If you later decide that you'd like to remove a job you submitted with `at`, all you do is use the `-r jobIDs` switch. Just replace `jobIDs` with the list of job identifiers that `at` gives you when you start the jobs. Since the `-l` option doesn't provide much detail, it can be difficult to know for sure which job needs to be removed if you use that `at` command heavily.

For this reason, we suggest that you either write down the job identifiers when `at` gives them to you or stagger execution by at least a minute and keep track of the time you submit them. This is the simpler approach, since it's easier to keep track of the time the job should run.

Suppose for a moment that we wanted to delete the second and fourth jobs listed in Figure A. To do so, we'd simply type

```
# at -r 846820200.a 846837000.a
```

Restricting the at command

Once you're familiar with the `at` command, you'll be able to submit massive jobs at times when you're not going to be around. However, if the machine performs some mission-critical tasks, you might not want users indiscriminately deferring computer-bound jobs to the middle of the night during a month-end closing or similar process.

The *at.deny* and *at.allow* files, located in the */usr/lib/cron* directory, let you control access to the `at` facility. By listing specific users in the *at.deny* file, you can prevent them from accessing the `at` command. If only a few users should be using it, you can put each user's name in the *at.allow* file.

The way this works is that the `at` command first checks for the existence of the *at.allow* file. If it exists, then the user must be listed to be able to use the `at` command. If the *at.allow* file doesn't exist, then the `at` command checks the *at.deny* file. If the user isn't listed there, then `at` grants the user permission to go ahead.

In a default installation, Solaris provides an *at.deny* file but no *at.allow* file. Thus, all user accounts should be able to access the `at` command by default. (The default *at.deny* file lists only a few system processes, such as smtp, that have no access to the `at` command.)

Conclusion

Solaris has existed for such a long time that hundreds of utilities are available to make your job simpler. In many cases, there's overlap between the functionality of different commands. In this article, we showed you the `at` command, which combines some of the flexibility of `crontab` with the ease of use of the `nohup` command. As you start to use the `at` command, you'll definitely want to check out its `man` page to learn about some of the features we haven't mentioned. ❖

A VERSATILE SCRIPTING LANGUAGE

An introduction to Perl

By Al Alexander

For several years, a unique programming language with a legion of loyal followers has grown in popularity in the UNIX world. At first, the growth was gradual and not widely known. But with the sudden explosion of the World Wide Web, this language has gained overnight fame as an outstanding CGI programming tool. This language is, of course, Perl, the Practical Extraction and Report Language.

Perl is possibly the best text-processing language I've ever worked with. It combines many of the best features of `sed`, `awk`, `shell`, and the C programming languages, which makes it very powerful for manipulating text streams and data files. However, Perl has grown beyond being a simple text-processing language. It now offers many features that make it very desirable as a general-purpose programming language.

Why Perl?

Perl is a powerful interpreted scripting language with many advanced functions and features. Its syntax is a mixture of the C, `awk`, `sed`, and shell programming languages. If you're comfortable working with any of these languages, learning Perl will be fairly simple.

Some of Perl's strengths are:

- Many built-in string, array, and list functions
- Many built-in UNIX system functions
- Built-in arithmetic functions
- The use of regular and associative arrays
- Simple dynamic creation and resizing of arrays
- Support for local and global variables
- Support for the `ed`/`sed` stream-editing command syntax
- Pattern-matching syntax compatible with `sed`, `awk`, and other UNIX utilities
- Easily controlled formatted output
- Built-in TCP/IP socket commands
- The availability of third-party libraries that have been created to link into databases from Oracle, Informix, Sybase, and others
- Freely available for Solaris, other UNIX variants, DOS, Windows, OS/2, and other platforms

Is Perl right for you?

I encourage you to look at Perl if you're beginning to find that the standard shell programming languages don't support your advanced programming needs. You should also look at Perl if you're consistently developing small-to-midsize applications with C; Perl lets you create these same applications much more easily.

Two other factors may sway you to try Perl. First, it's offered on many platforms and for many operating systems. So if you need to create programs that work in multiple locations, you may want to give it a try. Finally, Perl has become famous for being the Internet/CGI programming language of choice. If you're building a Web site, you may want to use Perl to take advantage of the wealth of code already available.

Programming constructs

Figure A shows a brief Perl program that demonstrates many Perl basics. Looking at this program, people are generally either intimidated by Perl's syntax or they quickly see how they can accomplish so much with so few programming statements. Don't be intimidated—I can explain it all very easily!

The program is actually fairly simple. It first runs the `ps -ef` command via an `open` statement. The final pipe symbol (`|`) forces the output of the `ps -ef` command into a pipe, accessible to the `open` statement. In the `open` statement, we assign the name `PSEF` to the file handle assigned to the `ps -ef` output stream. Now, whenever we want a line of results from the `ps -ef` command, we simply refer to the `PSEF` file handle.

If you want, you can open multiple files and command pipe streams in a program. Contrast this to a shell script, where it's somewhat difficult to read from a file.

The `while (<PSEF>)` syntax determines whether any information remains in the `PSEF` input stream. If so, it reads a line from it and places the resulting line in the `$_` variable. If 100 processes are running on your system, this `while` loop executes 100 times. (Actually, because the `ps -ef` output also includes a header as its first line, the loop executes 101 times, but we're ignoring the first line for the purposes of this example.) Inside the `while` loop, we'll process the `ps -ef` output stream one line at a time.

On the next line, the `chop($_)` command removes the last character (a newline in this case) from the variable `$_`. In this way, we can treat the input line as a series of fields separated by white space, without worrying about the problems a newline character can create. (White space is any combination of blanks, tabs, or newline characters.)

The next line shows another great feature of Perl: Here, we split the line read from `ps -ef` into eight output fields. We then assign these eight fields to to eight separate variables, beginning with `$uid`. The first argument to the `split` command tells `split` which pattern to use to separate the fields. The next argument tells `split` which string to break apart. The third argument tells `split` to break the information in `$_` into no more than eight output elements.

Figure A

```
#!/usr/bin/perl
open(PSEF, "ps -ef|");
while (<PSEF>) {
  chop($_);
  ($uid,$pid,$ppid,$c,$stime,$tty,$tmptime,$cmd) = split(' ',$_,8);
  ($min,$sec) = split(/:/,$tmptime,2);
  $time = 60 * $min + $sec;
  if ($time > 30) {
    print "PID $pid, owner $uid, time $time, cmd $cmd\n";
  }
}
close(PSEF);
```

This small program demonstrates many of Perl's powerful features.

In this example, we're using `$_` again, telling `split` to break apart the current line from the `ps -ef` command, with up to eight fields delimited by spaces. We limit the number of fields, because the last column in the output of `ps -ef` contains the command line the process is executing, which may contain embedded spaces. If we omitted the number of fields limit, it would throw off the `split` command and allow it to create nine or more output fields. Think about how much more you'd have to do in a C program or shell script to accomplish the same goal!

Notice everything that happens in this one line: `split` takes the input from the variable `$_` and breaks this input into eight separate fields. It assigns these eight output fields to eight different variables at one time. Then we assign the resulting values to the variables `$uid` through `$cmd`. From this one line, you can see how Perl's syntax is a great time- and space-saver.

In our sample program, we want to find the amount of CPU time each process consumes. Since the `ps -ef` command prints the CPU time in min:sec format, we must do a little more work.

The next line of the program splits the variable `$tmptime` into two new variables named `$min` and `$sec`. In this line, we tell the `split` command that the input fields are delimited by a colon (:) character. The result is that `$min` and `$sec` hold the number of CPU minutes and seconds accumulated by the process. If a process has accumulated 10 minutes and 15 seconds of CPU time, `$tmptime` will contain "10:15". After using `split`, `$min` holds 10 and `$sec` holds 15.

The next line demonstrates that math is very easy in Perl, as we combine `$min` and `$sec` to create the new variable `$time`. Now `$time` contains the accumulated runtime of the current process in seconds.

The next line of the program introduces the `if` statement. As with C and the C shell, the condition is enclosed in parenthesis. After the `if` statement, you're required to have a block that encloses the statements you want to conditionally execute.

For our example, if the variable `$time` is greater than 30 seconds, Perl prints the process ID (PID), User ID (UID), amount of CPU time consumed, and the UNIX command line to the standard output. (Note that Perl also provides a `printf` function, which we could have used. While the `printf` function provides greater control of the output, it's less efficient. Rather than go into the `printf` function, we'll illustrate report formats in the next section.)

The right curly brace on the next line marks the end of the `if` statement, and the next right curly brace marks the end of the `while` statement.

The last line of the program closes the connection to the `ps -ef` pipe. While this isn't necessary in this case, you'll generally want to close any file or pipeline you open.

In summary, as this program runs, it reads each line of the `ps -ef` output and prints each process that has an accumulated run time greater than 30 seconds. When the program can read no more lines of input from the `ps -ef` pipe, it exits. A typical run on our machine looks like this:

```
PID 418, user root, time 1566, cmd ./POV-Ray
PID 466, user root, time 58, cmd ./dbSort
PID 474, user root, time 37, cmd ./a.out 99999
```

Getting even fancier

Perl's motto is "There's more than one way to do it." Figure B shows another way to solve the same problem. In this case, we're going to use some of Perl's built-in report-formatting features and demonstrate a couple of other features in Perl.

Figure B

```
#!/usr/local/bin/perl
format top =
     CPU hogs
PID    Owner      CPU sec  Command
-----  ---------  -------  --------------------
.
format STDOUT =
@>>>>  @<<<<<<<<  @>>>>>>  @<<<<<<<<<<<<<<<<<<<
$ps[1],$ps[0],  $time,  $ps[7]
.
open(PSEF, "ps -ef|");
while (<PSEF>) {
  chop;
  @ps = split(' ',$_,8);
  ($min,$sec) = split(/:/,$ps[6],2);
  $time = 60 * $min + $sec;
  write if ($time > 30);
}
close(PSEF);
```

Here, we've changed the program to illustrate arrays and Perl's report-formatting features.

In this version of our script, the first thing you'll probably notice are the blue lines. These lines define the format of our report. The `format top` statement tells Perl what to print at

the top of each page. The `format top` statement ends when it comes to a line containing a single period. As you can see, we're just telling Perl to print the report name and some column headings.

The `format STDOUT` statement tells Perl how to print each line. Here, the first line defines four fields (starting with the `@` symbol). The `<` and `>` characters following the `@` symbol tell Perl how wide the field is and how to align the data. Perl computes the field width by simply counting the characters used in the field definition. In the first field, the `@` character is followed by four `>` symbols, for a total of five characters. You can use three alignment characters: `>` tells Perl to right-justify the field, `<` means to left-justify the field, and `|` means to center the field. In all cases, Perl pads out the field with spaces.

As part of the `format` statement, you also can assign the variables associated with each field. As you can see, the third field in our `format STDOUT` statement is associated with the `$time` variable. Whenever Perl executes a `write` statement, it formats the specified variables in the fields, then prints the record.

The next thing you might notice is that we've changed the `chop($_)` statement in Figure A to `chop`. We can do so because, as we mentioned earlier, the `$_` variable is the default variable for many operations in Perl. If you don't tell Perl what variable you want to `chop`, it assumes that you want to `chop` the `$_` variable.

The very next statement is different as well. Here, we're using an array to hold the results of the `split` operation. So instead of providing a list of variables to hold the `split` results, we specify an array. When Perl `split`s the input record, the first field goes into slot 0 of the `ps` array, the next field goes into slot 1, etc.

Please note that we must tell Perl that we're using an array by using an `@` character at the beginning of the array instead of the usual `$`. In Perl, whenever you use a variable, you must tell Perl how you're planning to use it. When you use it as a scalar value, the default case, you use `$`, just as you do in shell programming. When you want to use the variable as an array, you use the `@` character. Now here's the tricky part: If you want to get a scalar (i.e., a single value) out of your array, you use the `$` symbol and enclose the subscript in brackets. The next line demonstrates this, as we `split` the CPU time variable into the `$min` and `$sec` variables. As you can see below, we use `$ps[6]` to access the seventh value in our `@ps` array:

```
($min,$sec) = split(/:/,$ps[6],2);
```

Remember earlier when we said that an `if` statement must be followed by a block of statements surrounded by curly braces? Well, that's not strictly true. In Perl, just like in C, you *can* follow an `if` statement with a block of statements in curly braces. In C, you can use a single statement instead, like this:

```
if ($time > 30)
  statement;
```

However, that's illegal in Perl, although Perl *does* provide a shorthand notation for conditionally executing a single statement: You may follow the statement with your `if` clause. Please note that the `if` clause must precede the semicolon. This example, from Figure B, shows how we write our results to the report:

```
write if ($time > 30);
```

As you can see, it executes the `write` statement *if $time is greater than 30*. Otherwise, Perl ignores the statement.

Our new version of the report prints the data in a much nicer format. When you run it, you'll see a report much like this:

```
     CPU hogs
 PID  Owner      CPU sec  Command
----- ---------  -------  ----------------------
  474 root           519  ./a.out 99999
  418 root          1937  ./POV-Ray render1
  466 root           523  ./dbSort
```

Conclusion

As a general-purpose scripting language, Perl is a great improvement over shell languages. Perl provides support for local variables, numeric and associative arrays, built-in arithmetic functions, low-level I/O functions, and dozens of functions to manipulate strings, lists, and arrays. Perl further offers support for TCP/IP sockets programming and other advanced features.

Perl is also an easier language to use than C for creating most system-administration utilities and CGI programs. Perl offers many of the best features of the C language, but requires fewer lines of programming and has no need for a compiler.

As an added bonus, the newest version of Perl, v5.003, now offers support for references (similar to C pointers), complex data structures, and object-oriented programming—not bad for a shell programming replacement. You can obtain Perl from many places on the Internet or through CD-ROMs from varied vendors. You can even get the source code from *http://www.perl.com/perl*. ❖

Using associative arrays in Perl

By Al Alexander

One of the great features of the Perl language is its support of associative arrays. Unlike normal arrays, whose subscripts can only be integers, the subscripts of associative arrays are text strings. This may not sound like much yet, but we can use associative arrays to create fairly complex data structures with Perl.

Since associative arrays add so much power to Perl programming, you *must* become proficient with them to be a good Perl programmer. In this article, we'll describe associative arrays, and we'll show you how to use them in your own programs. Along the way, we'll also explain how to access all the variables in your environment.

What are associative arrays?

No doubt you're familiar with standard arrays. They're simply containers of values in which each value is identified by a key value from zero to *n*, where *n* is the number of items in the array.

An associative array is very similar, except that instead of using a number from zero to *n* to refer to the entries, you can give each entry a name. These names are strings that are *associated* with the value in the array, hence the name.

As we mention in the article "An Introduction to Perl," when you refer to a scalar value, you precede it with `$`, and when you refer to an array, you precede the array name with `@`, and enclose the key in brackets. As an example, here are two lines of code. The first line copies the array `fred` into the array `george`. The second line reads the third value from the array `george` into the scalar variable `ethel`:

```
@george = @fred;
$ethel = $george[2];
```

For an associative array, you use the `%` character to specify the array name, and you use curly braces to select an element by name. The next two lines of code copy the associative array named `lucille` to `mary`. Next, we select the value in `mary` associated with the string `joe` and place it in the variable named `garage`.

```
%mary = %lucille;
$garage = $mary{'joe'};
```

It's said that associative arrays use fancier brackets than normal arrays because associative arrays are fancier than normal arrays. While I don't know if this statement is 100 percent accurate, I do find it to be an effective way of remembering the proper subscripting syntax when using Perl arrays.

Accessing the environment

When you start a program, Perl initializes an associative array for you: The associative array named `%ENV` holds the values of all shell environment variables. Figure A shows a complete Perl program that uses associative arrays. It's an improved version of the traditional "Hello, world" program often used to demonstrate a new programming language.

Figure A

```
#!/usr/bin/perl
$userName = $ENV{'LOGNAME'};
print "Hello, $userName\n";
```

This program can greet a user by name by using the %ENV associative array to locate the user's login name.

The login process stores your user name in the `LOGNAME` environment variable. This brief Perl program takes advantage of this fact to obtain your user name from the `%ENV` associative array.

Iterating over an associative array

In a normal array, it's easy to scan through the array and process all the array items: Simply count from zero to *n* and process the corresponding array item. But just how do you do it with an associative array?

To answer this problem, Perl provides the `keys` command. This command creates a new array, which contains all the keys (i.e., subscripts) in the associative array. Then you can use the `foreach` command to iterate over the keys array.

As an example, let's display all the environment variables. The program shown in Figure B displays a sorted list of all the environment variables.

In this example, the keys of the associative array `%ENV` are the names of the environment variables. Back in Figure A, the key (or

subscript) for the %ENV array was LOGNAME, and the value of the array element $ENV{'LOGNAME'} was fred. Suppose that Table A shows all the environment variables you have set when you run the program.

Figure B

```
#!/usr/bin/perl
foreach (sort keys %ENV) {
  print "$_ = $ENV{$_}\n";
}
```

This program displays all the environment variables.

When Perl executes the keys command within the foreach loop, it returns the subscript names LOGNAME, HOME, SHELL, and EDITOR. It does not return the values of each element.

The sort command (preceding the keys command) sorts the resulting array of keys. The foreach command then iterates over the sorted list of keys. This is how the program prints all the environment variables in order. Given the keys shown in Table A, the sorted output would leave the keys in the following order: EDITOR, HOME, LOGNAME, and SHELL.

The foreach statement creates a loop that you can read as "for each element in the list 'EDITOR, HOME, LOGNAME, and SHELL', do everything enclosed in the following curly braces." This tells Perl to execute the print statement for each key in the %ENV array.

As mentioned in "An Introduction to Perl," the $_ variable can have many meanings, depending on the context. In the foreach statement, since we don't explicitly provide a loop variable, Perl assigns the value of the current key to it for each iteration of the loop. The first time through the loop, the print command

```
print "$_ = $ENV{$_}\n";
```

substitutes EDITOR for the value of $_ and runs the command

```
print "EDITOR = $ENV{'EDITOR'}\n";
```

Table A

Perl statement	Key	Value
$ENV{'LOGNAME'} = "fred";	LOGNAME	fred
$ENV{'HOME'} = "/home/fred";	HOME	*/home/fred*
$ENV{'SHELL'} = "/bin/ksh";	SHELL	*/bin/ksh*
$ENV{'EDITOR'} = "vi";	EDITOR	vi

Using the %ENV environment variable, it's easy to access environment variables by name.

which results in the output

```
EDITOR = vi
```

The foreach loop then continues its processing until every key from the array %ENV has been processed.

Uses of associative arrays

Associative arrays can simplify your programs in several ways. One way is to allow you to track less data. When you use standard arrays, you're limited to subscripts of zero through *n*, so you often have to create an extra array that will translate the zero to *n* values to the values you really care about. With an associative array, you can use the interesting variables directly to index the data array.

One benefit is that you can often model complex data using fewer arrays. Thus, it's easier for you to visualize what's going on, and you make fewer program errors.

The cost of an associative array is that Perl has to perform a little more work to access a value through an association compared to a standard array's subscript. However, for many projects, the extra efficiency gained by a standard array isn't worth the added extra work and complexity.

Conclusion

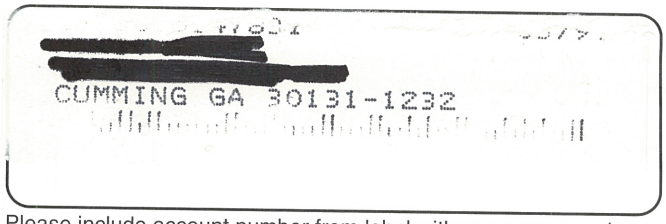
The associative arrays in Perl can make programming much simpler. We've shown you how easy it is to access the environment using Perl's associative arrays. Instead of having to deal with many different variable names, we simply work with one array that holds the contents of the separate variables. ❖

Statement of Ownership, Management and Circulation (Required by 39 U.S.C. 3685) 1. Publication Title: **Inside Solaris**. 2. Publication number: **0013674**. 3. Filing date: October 3, 1996. 4. Issue Frequency: **Monthly**. 5. No. of Issues Published Annually: **12**. 6. Annual Subscription Price: **$115 ($135 Foreign)**. 7. Complete Mailing Address of Known Office of Publication: The Cobb Group, 9420 Bunsen Parkway, Louisville, KY 40220. 8. Complete Mailing Address of the Headquarters of General Business Offices of the Publisher (Not printer): The Cobb Group, 9420 Bunsen Parkway, Louisville, KY 40220. 9. Full Names and Complete Mailing Address of Publisher, Editor, and Managing Editor: Publisher, John Jenkins, The Cobb Group, 9420 Bunsen Parkway, Louisville, KY 40220; Editor, **Marco Mason**, The Cobb Group, 9420 Bunsen Parkway, Louisville, KY 40220; Managing Editor, Linda Baughman, The Cobb Group, 9420 Bunsen Parkway, Louisville, KY 40220. 10. Owner: Ziff Davis Publishing Company, 1 Park Avenue, New York, NY 10016; Softbank Holdings Inc., 10 Langley Road, Suite 403, Newton Center, MA 02159. 11. Known Bondholders, Mortgagees, and Other Security Holders Owning or Holding 1 Percent or More of Total Amount of Bonds, Mortgages or Other Securities: None. 13. Title of Publication: **Inside Solaris**. 14. Issue Date for Circulation Data Below: November 1996. 15. Extent and Nature of Circulation—A. Total No. Copies (Net Press Run): Average No. Copies Each Issue During Preceding 12 Months, **4,281**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **4,776**. B. Paid and/or Requested Circulation—1. Sales through dealers and carriers, street vendors and counter sales (Not mailed): Average No. Copies Each Issue During the Preceding 12 Months, **65**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **78**. 2. Paid or Requested Mail Subscriptions: Average No. Copies Each Issue During the Preceding 12 Months, **3,538**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **3,164**. C. Total Paid and/or Requested Circulation (sum of 15b(1) and 15b(2)): Average No. Copies Each Issue During the Preceding 12 Months, **3,603**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **3,242**. D. Free Distribution by Mail (Samples, complimentary, and other free): Average No. Copies Each Issue During the Preceding 12 Months, **20**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **18**. E. Free Distribution Outside the Mail (Carriers or other means): Average No. Copies Each Issue During the Preceding 12 Months, **0**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **0**. F. Total Free Distribution (sum of 15d and 15e): Average No. Copies Each Issue During the Preceding 12 Months, **20**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **18**. G. Total Distribution (Sum of 15c and 15f): Average No. Copies Each Issue During the Preceding 12 Months, **3,623**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **3,260**. H. Copies Not Distributed. 1. Office use, left over, unaccounted, spoiled after printing: Average No. Copies Each Issue During the Preceding 12 Months, **658**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **1,516**. 2. Return from News Agents: Average No. Copies Each Issue During the Preceding 12 Months, **0**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **0**. I. Total (Sum of 15g, 15h(1), and 15h(2)): Average No. Copies Each Issue During the Preceding 12 Months, **4,281**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **4,776**. Percent Paid and/or Requested Circulation (15c/15g x 100): Average No. Copies Each Issue During the Preceding 12 Months, **99.44%**; Actual No. Copies of Single Issue Published Nearest to Filing Date, **99.45%**. This Statement of Ownership will be printed in the January issue of this publication. I certify that all information furnished on this form is true and complete. I understand that anyone who furnishes false or misleading information on this form or who omits material or information requested on the form may be subject to criminal sanctions (including fines and imprisonment) and/or civil sanctions (including multiple damages and civil penalties). Director-Fulfillment Operations.

PERIODICALS MAIL

SunSoft Technical Support

(800) 786-7638



Please include account number from label with any correspondence.

TIME-SAVING PRODUCT

Summertime...in the dead of winter

If you always find yourself searching the Internet for programs, then configuring and compiling them to run under Solaris, you can save yourself some time. Micromata created the Summertime '96 for the Solaris™ Environment CD-ROM. This is a package of all the popular shareware and freeware tools, already compiled and packaged for Solaris 2.5. Micromata makes a CD-ROM for both the Sparc and x86 platforms.

You'll find such things as the GNU tools (gcc, emacs, etc.), Perl, TCL, TeX, alternative shells, games, security tools, Web servers, Web browsers, the JDK, IRC, database servers, mail tools, and many other items—over 270 applications, all told.

While all these are available in various places on the Internet, this CD-ROM puts them in a convenient package: No need to locate, download, or archive them! Just load whatever you want from the CD-ROM.

If you have a slow pipeline to the Internet or just prefer the convenience of having all these popular programs on a single CD-ROM, then you'll want to investigate this package. In the United States, Canada, and Mexico, call EIS Computers, Inc. at (800) 351-4608 or visit its Summertime '96 Web page at *http://www.eis.com/summertime*. Elsewhere, visit Micromata's Web page at *http:/www.micromata.com*, or call +49-(0)-5624-925230. ❖

LETTERS

Revisiting the xterm title bar

I'd like to react to the article "Changing the Title Bar of an xterm Window" in the November issue. I've been using `xterm` for a while, and all I have to do to change the title or icon text on my window is use the `-T` or `-n` option of the `xterm` command. Isn't this method easier?

Wim Wauterickx
via the Internet

Wim, for most cases you're right. If you know what title and icon text you want when you start your `xterm` window, the `-T` and `-n` options are definitely the way to go. However, if you want to *change* the title and/or icon text, in a shell script, for example, you must use the technique we've described.

Suppose you have a script that you run in the background to monitor your system. Since you don't want to see the monitor process running continuously, you minimize the window and give it an icon name of "monitor." Now suppose your script finds a problem. You could have it change the icon name to "PROBLEM FOUND," which you would notice on your desktop. ❖