February 1998 • $11.50
Vol. 4 No. 2

ZIFF-DAVIS
THE COBB GROUP

INSIDE SOLARIS™
Tips & techniques for users of SunSoft Solaris

IN THIS ISSUE

Visit our Web site at http://www.cobb.com/sun

Running Solaris commands from Java applications

by Alvin J. Alexander

As a Solaris system administrator, developer, or end user, you may have looked at Java by now and decided that it's an environment worth investigating. The combination of a relatively simple object-oriented language with a platform-independent philosophy and browser integration has taken the industry by storm. More than just a good programming language, Java is a programming environment around which new industries—Java Bean components, Smart Cards, and Network Computers to name a few—are being built.

As I dug into Java from an administrator's perspective, my first questions centered around running Solaris commands from Java applications. If you're interested in how you might use Java programs for system administration, then this article is for you. In this article, we'll tackle the first important skill—running Solaris commands from within your custom Java applications.

A first example

Beginning with the JDK 1.0, Java has provided a way to run system commands from Java applications for all operating system environments. For Solaris administrators interested in writing Java programs that run Solaris commands, the Java classes named `Process` and `Runtime` give the Java program access to some of the pieces underlying the operating system.

The `Runtime` class provides an `exec()` method that lets the Java programmer run commands. You simply call the `exec()` method of the `Runtime` class, supplying the command string you want to run. The `exec()` method then creates a `Process` object that runs the specified command.

Listing A on page 2 shows a minimal Java program, named `RunCommand`, that runs a Solaris command (`ps -ef`) from a Java application. This code is minimized so much, in fact, that it ignores the output of the `ps` command after it's run! We'll take care of that problem a little later, but first let's dig into the code in Listing A to see what's happening.

Our program's first statement is the `import` statement, which specifies classes you want to use in your program. The `import` statement we used tells Java that we want access to *all* the classes in the `java.io` package. (We could choose only a specific class, but it's often simpler to use the whole package.)

The classes that comprise the `java.lang` package are used heavily in Java programs, so you needn't import the `java.lang` package: It's implicitly imported for you. Please

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
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@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: cobb_customer_relations@zd.com

Staff
Editor-in-Chief ..... Marco C. Mason
Contributing Editors ..... Al Alexander
Print Designer ..... Margueriete Winburn
Editors ..... Karen S. Shields
Joan McKim
Michael E. Jones
Publications Coordinator ..... Linda Recktenwald
Managing Author ..... Eddie Tolle
Product Group Manager ..... Michael Stephens
Circulation Manager ..... Mike Schroeder
Publisher ..... Jon Pyles
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.

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

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

note also that you can still access classes that you don't explicitly import. However, when you use them, you must fully qualify their names. For example, the `IOException` class is part of the `java.io` package. If we didn't include the `java.io` package, we'd have to use `java.io.IOException` each time we wanted to use this class. Therefore, the `import` statement merely gives us a shorthand notation for writing our code.

The next line of code declares our new class named `RunCommand`. You must put your code inside a class, because *all* Java code exists within classes. Of course, you needn't name your class `RunCommand`—you can name it anything you want. However, you *must* declare your Java program inside a class.

If you're new to object-oriented programming, declaring a class in a small program like this doesn't seem to make much sense. But once a program becomes just a little larger, the class approach of the object-oriented model becomes much more intuitive and elegant.

Now, how does a Java program know where to start? Well, if you've had any exposure at all to the C or C++ programming languages, you know that program execution starts in the `main()` function. The same is true in Java. As you can see, the rest of our program is inside the curly brackets of the `main()` routine. When you write Java applications (as opposed to applets), you'll see this same declaration for the `main()` function again and again.

Within the `main()` routine, you encounter Java's `try/catch` syntax. This syntax is Java's way of trapping errors, or *exceptions*, that may occur while a program is running. I like to think of this syntax as meaning "go ahead and *try* this block of commands, but if an error happens, stop running those commands, and run the commands in the *catch* block instead." So, if an exception occurs, Java *catches* the error and continues to run, instead of failing completely. As you work with Java, you'll find this syntax a powerful way of handling all sorts of possible errors in your Java programs.

If we apply this thought process to our program in **Listing A**, we see that the Java interpreter will attempt to run the three commands inside the `try` block. Specifically, we'll try to create a new `Process` in which to run the `ps -ef` command. Next, the program prints a message to inform us that the `ps -ef` command ran, and it'll `exit`. However, if any of those commands fail because of an I/O exception, Java will start executing the code in the `catch` block—running the `print` statement and `System.exit(-1)` command instead. If no error occurs, the program ignores the `catch` block entirely.

Listing A: *RunCmd*

```
import java.io.*;

public class RunCommand {

    public static void main(String args[]) {

        try {
            Process psef = Runtime.getRuntime().exec("ps -ef");
            System.out.println("ran the ps -ef command");
            System.exit(0);
        }
        catch (IOException e) {
            System.out.println("Error occurred trying to run ps -ef");
            System.exit(-1);
        }
    }
}
```

The method `System.out.println()` is similar to an `echo` statement in the C, Bourne, or Korn shells: It simply writes the text you specify to standard output. The `System.exit(x)` method is Java's way of returning an exit status of *x* to the caller. It's the same as the `exit` statement in the Bourne shell.

Compiling and running your Java program

After you've written your code, running a Java program is more like running a C/C++ program than running a shell script. Unlike a shell script, you can't just save this code to a file, change its permissions, then run it. With Java, you save the file, compile it, and run the compiled code through the Java interpreter.

First, you should save the code in a file named *RunCommand.java*. Next, you must compile this source code file into a Java byte-code *.class* file. You do this by running the Java compiler, named *javac*, as follows:

```
$ javac RunCommand.java
```

This command creates a class file that's named *RunCommand.class*, which contains your platform-independent Java byte-code. Once you've created this file, you can run it on any platform that has a Java interpreter.

Please note that when you name your source file, you must name it the same as your class name, with the addition of the *.java* extension. Since the name of our new class is `RunCommand`, we name the source file *RunCommand.java*. Then, when javac compiles *RunCommand.java*, it creates the file *RunCommand.class*. This is important, because when you tell you the `java` command to run the `main()` function in a specific class, `java` expects to find the executable code in a file named after the class name with the extension *.class*.

When you want to run this program, you run it through the Java interpreter, `java`. In our case, after compiling the code in Listing A, run the code through the Java interpreter as follows:

```
$ java RunCommand
```

This command loads the Java byte-code interpreter, telling it which class to start with. So `java` loads the *RunCommand.class* file and begins executing it at the function named `main()`. If everything ran successfully, the output from this program should be

```
ran the ps -ef command
```

Although this result isn't very useful, you've just run the `ps -ef` command from a Java program. Once you learn how to read the output from the `ps -ef` command and print that output to your terminal, you're ready to add a variety of Solaris commands to all of your Java applications.

Reading the output of the Solaris command

In order to start the `ps -ef` command in our program, we created a `Process` object. When you create a new `Process` object, Java allows you access to the standard input, output, and error streams of the new `Process`, so you can communicate with the running program.

In our example, we've named our `Process` object `psef`. All `Process` objects have a built-in method named `getInputStream()` that connects the standard output of the running process into the specified input stream coming into your Java application. As an administrator, you're comfortable with the Solaris concepts of standard input, standard output, and standard error, so you should feel right at home with the idea of an input stream.

For a programmer using Java, it's easiest to convert a raw input stream like this into something far more useful, specifically a `DataInputStream`. Such a raw input stream that is provided by the `Process` object's `getInputStream()` method isn't very convenient, whereas a `DataInputStream` is easier to work with.

Therefore, our first step toward reading the output of the `ps -ef` command is to convert the `Process` object's raw input stream into a `DataInputStream` object. We can easily accomplish this with the following Java code:

```
DataInputStream dis = new DataInputStream(
  psef.getInputStream() );
```

To make our Java application run faster, we'll also buffer the input stream by using a `BufferedInputStream` class. In our tests, buffering the input stream improves the read performance almost ten-fold, making it well worth any additional code. Instead of adding an extra line of code to the program, you can simply insert the `BufferedInputStream` method call between the raw `psef` object and the `DataInputStream` method call, like this:

```
DataInputStream dis = new DataInputStream(
    new BufferedInputStream(
        psef.getInputStream() ) );
```

Now, any time you want to read the input stream, simply read from the object `dis`, using the methods of the `DataInputStream` class. For instance, after invoking the commands above, it's easy to read from `dis` using the `DataInputStream` object's `readLine()` method. To read the first line of input from the `ps -ef` command, you'll use this code:

```
String s;
s = dis.readLine();
```

This command gets the first line of input from the `DataInputStream` object `dis` and puts that data into the `String` variable named `s`.

In order to read every line of output of the `ps -ef` command, you should put the `readLine()` method into a `while` loop. In the `while` loop, you'll keep reading the output of the `ps -ef` command until the `readLine()` method returns a null value. You'll get a null value when there's no more output from the `ps -ef` command. In Java, you'll create the `while` loop like this:

```
String s;
while ((s = dis.readLine()) != null) {
    //  process the string "s" here...
    System.out.println(s);
}
```

Putting it all together

Listing B shows a new version of our program, named `RunPScommand`, where we invoke the `ps -ef` command and attach a `DataInputStream` object to the output of the `ps -ef` command. We read each line from the `DataInputStream` object with the `readLine()` method and print the value of the string `s` to the standard output.

Listing B: *RunPScommand*

```
import java.io.*;
public class RunPScommand {
   public static void main(String args[]) {
      try {
          Process psef = Runtime.getRuntime().exec("ps -ef");
          DataInputStream dis = new DataInputStream(
              new BufferedInputStream(
                  psef.getInputStream() ) );
          while ((s = dis.readLine()) != null) {
              System.out.println(s);
          }
          System.exit(0);
      }
      catch (IOException e) {
          System.out.println("Error:  Tried to run ps -ef");
          System.exit(-1);
      }
   }
}
```

To run this program, we'll perform the same steps we did previously. Simply save the changes to the *RunPScommand.java* file, compile the Java source code into a class module, then run it, like so:

```
$ javac RunPScommand.java
$ java RunPScommand
ran the ps -ef command
 UID   PID  PPID  C    STIME TTY  TIME CMD
root     0     0  0  Nov 01 ?    0:00 sched
root     1     0  0  Nov 01 ?    0:01 /etc/init
-
root     2     0  0  Nov 01 ?    0:00 pageout
root     3     0  0  Nov 01 ?    2:12 fsflush
```

Some other thoughts

The methods demonstrated in this article let you run Solaris commands from Java *applications* on Solaris workstations. You shouldn't confuse this action with running commands from Java applets in Internet browsers, which is a bit different. Because of the security model that's used for Java applets running in browsers, we must tackle a few more issues, which we'll do in a future article.

Please note also that some fundamental changes in Java have occurred between the versions implemented in the JDK version 1.0 and version 1.1. For example, the `readLine()` method is now deprecated (i.e., works in version 1.1, but you shouldn't use it in new programs because the next version of Java probably won't support it.) In a nutshell, this means that although these classes will still be included in the JDK, you should begin using the newer classes as soon as possible. Since a mixture of JDK 1.0 and 1.1 users are still out there, we have supplied the code that works with both developer kits.

Conclusion

There's no question that using Java to run Solaris commands is more difficult than running the same commands inside of a shell program. But remember, we've only just begun. By developing your own custom classes, you can make the entire process as easy as this:

```
DataInputStream dis = new SolarisCommand
➥("ps -ef");
```

Also, if you want to run Solaris commands from a graphical interface, Java is rapidly becoming the simplest language to use to create cross-platform GUI applications. Finally, this same approach is your first step down the road of running Solaris commands from Internet browsers. ❖

Using the keyboard to switch between CDE workspaces

When I'm working on a large project, my desktop is usually cluttered with `xemacs` and `dtterm` windows, arranged to fill all the available screen space. Until recently, I've always had to make sure that a corner of the CDE Front Panel showed somewhere, so that I could click on it and switch to another workspace to take notes for a phone call or some other task.

However, arranging your windows so that you can always see a corner of the Front Panel can be tiresome, especially when you want to use that last little bit of space to see another line or two of code. Imagine my delight when I found out how to switch between CDE workspaces using keyboard commands!

Configuring the keyboard

By default, the CDE doesn't provide this functionality. However, two architectural decisions make it possible. First, the CDE was written with an event-passing architecture. Second, the CDE also allows you to send these events to various CDE components very easily. Thus, if you want to spend some time working with and learning `dtksh` (a version of the Korn shell that has the ability to interact with the CDE and X windows), you can do some pretty incredible things with CDE.

Fortunately, we don't have to go to such lengths. Since `dtwm` (the Desktop Window Manager, upon which CDE is built) is the focal point of all this CDE activity, it's built with flexibility and events in mind. Thus, you can tie particular events to specific actions, such as clicking on a window or pressing a key. Even better, you can tweak `dtwm`'s behaviour with its configuration file. Now, we can divide our task into four simple steps:

- Create a configuration file for `dtwm`
- Find the events we want to use
- Select keys to switch our workspaces
- Update our configuration file

Where do we start?

Our first task is to create the configuration file for `dtwm`. We begin the configuration by reading the `man` page for `dtwm`, which has plenty of very interesting information. For our purposes, the most useful part is how `dtwm` searches for its configuration file, as described in the `configFile` resource. If the `configFile` resource specifies a file, then `dtwm` attempts to use that file as a configuration file. Failing this, `dtwm` checks the following files, in order, until it finds one:

```
$HOME/.dt/$LANG/dtwmrc
$HOME/.dt/dtwmrc
/etc/dt/config/$LANG/sys.dtwmrc
/etc/dt/config/sys.dtwmrc
/usr/dt/config/$LANG/sys.dtwmrc
/usr/dt/config/sys.dtwmrc
```

Thus, `dtwm` first checks whether the user has a configuration file in either of the standard locations. Since `$LANG` is the language you log in with, you can have a different configuration for each language you use. If you use only one language, you can put your customizations in *$HOME/.dt/dtwmrc*). If the user doesn't have a configuration file, `dtwm` then checks for a customized, system-wide configuration file (for the specific language or a default one). If this search also fails, then `dtwm` uses the standard out-of-the-box configuration file.

Please note that `dtwm` stops searching for a configuration file when it finds the first one. So if you want to make a minor change to the current configuration, you need to include in your configuration file all the information that `dtwm` is presently using. To do this, simply locate the configuration file you're currently using when you log in and copy it to the location *$HOME/.dt/dtwmrc*.

```
$ cd $HOME
$ ls .dt/$LANG/dtwmrc
.dt/C/dtwmrc: No such file or directory
$ ls .dt/dtwmrc
.dt/dtwmrc: No such file or directory
$ ls /etc/dt/config/$LANG/sys.dtwmrc
/etc/dt/config/C/sys.dtwmrc: No such file or directory
$ ls /etc/dt/config/sys.dtwmrc
/etc/dt/config/sys.dtwmrc: No such file or directory
$ ls /usr/dt/config/$LANG/sys.dtwmrc
/usr/dt/config/C/sys.dtwmrc
$ cp /usr/dt/config/C/sys.dtwmrc .dt/dtwmrc
```

Now we've accomplished our first task, and we have our own configuration file. Since we're reading the man pages for dtwm and dtwmrc anyway, the second task is well underway. We must simply determine which events will let us switch workspaces and how to use the events.

In the dtwm man page, we discover our first clue in this statement:

> The following types of resources can be described in the dtwm resource description file:
>
> Buttons — Window manager functions can be bound (associated) with button events.
>
> Keys — Window manager functions can be bound (associated) with key press events.
>
> Menus — Menu panes can be used for the window menu and other menus posted with key bindings and button bindings.
>
> The dtwm resource description file is described in dtwmrc(4).

We then read man page dtwmrc(4) and hit the jackpot: There's a description of more than 50 event functions we can use. We can bind them to mouse buttons, key presses, and menus. After searching through the function list, we find three that do exactly what we want:

```
f.goto_workspace workspace
f.next_workspace
f.prev_workspace
```

Figure A

```
###
#
# Key Bindings Description
#
###

Keys DtKeyBindings
{
# Alt<Key>Menu       root!icon!window        f.toggle_frontpanel
  Shift<Key>Escape   icon!window             f.post_wmenu
  Alt<Key>space      icon!window             f.post_wmenu
  Alt<Key>Tab        root!icon!window        f.next_key
  Alt Shift<Key>Tab  root!icon!window        f.prev_key
  Alt<Key>Escape     root!icon!window        f.next_key
  *  *  *
}
```

This section of the dtwmrc file tells dtwm what to do with specific key combinations.

The first of these, f.goto_workspace accepts a single parameter: the name of the workspace you want to use. The f.next_workspace event function takes you to the next workspace in the list. If you're already at the last workspace, it wraps around back to the first one. As you'd expect, the f.prev_workspace event function takes you to the previous workspace or to the last workspace if you're presently at the first one.

Which keys do we want to use?

Next, we must choose which keys we want to use to switch workspaces. At the risk of getting ahead of ourselves, we'll point out that you shouldn't select key combinations that you'll use frequently in your applications or that already have a meaning, unless you're content to remove the previous meaning of those particular key combinations.

For our purposes, we chose [Alt][F1] to switch to the first workspace, [Alt][F2] for the second, and so on. We also decided to use [Alt][PageUp] and [Alt][PageDown] to cycle forward and backward through the workspaces. This way, we can try out all three of the event functions.

Modifying the configuration file

Now for the critical part: We must modify the configuration file. If we're not careful when we modify this file, the CDE might act very strangely.

At this point, we know what we want to do, but we don't know how to do it. Specifically, how do we bind the key presses to our new event functions? Luckily, the standard */usr/dt/config/C/sys.dtwmrc* file has plenty of customizations in it, so we can use them as examples. Browsing through the file, we come to the section that binds keys to events, part of which is shown in Figure A.

Take a look at the line in blue. The first part is obvious: When the user presses the space key while holding down the [Alt] key, dtwm is supposed to do something.

A bit more spelunking in the man page for dtwmrc shows that the second part tells when the key has a special meaning. In this case, it's active when an icon or window is selected. The last part is pretty intuitive: It's the action we're binding to the key.

You'll notice that some of the other key bindings have root!icon!window, instead of only icon!window. This difference simply means that

the key will work when the root window (i.e., background) is active. (This situation usually occurs when there's no active icon or window, so in effect, the key combination should almost always work.) Since we want our key combinations to work all the time, we'll have to use `root|icon|window`.

Now, we must figure out the names of the keys. We can make a pretty good guess, so we'll use F1..F4, PageUp, and PageDown. Putting this together, we come up with the lines shown in Figure B. Now, we simply add these new key bindings to the `DtKeyBindings` section of our *dtwmrc* file.

Figure B

```
Alt <Key>F1        root|icon|window  f.goto_workspace WebEdit
Alt <Key>F2        root|icon|window  f.goto_workspace SysAdmin
Alt <Key>F3        root|icon|window  f.goto_workspace RC10
Alt <Key>F4        root|icon|window  f.goto_workspace Ford DataMyter
Alt <Key>PageUp    root|icon|window  f.prev_workspace
Alt <Key>PageDown  root|icon|window  f.next_workspace
```

Adding these lines to the `DtKeyBindings` section should allow us to easily switch among the workspaces.

Well, how does it work?

Now we must log out of CDE and log back in, so that `dtwm` will read our customized *dtwmrc* file, instead of the one it was previously using. We hold our breath and press [Alt]F3. It works! Unfortunately, when we try the [Alt][PageUp] and [Alt][PageDown] combinations, they don't work at all. That's not too surprising, since we just guessed at the key names.

Digging through the *dtwmrc* `man` page again, we find that the key names should match those found in the */usr/openwin/include/X11/keysymdef.h* file, but with the `XK_` prefix omitted. Thus, we should've used `Page_Up` and `Page_Down`.

Please note: This file is only on your system if you installed the XWindows include files (package SUNWxwinc) on your system.

While thinking about this problem, we found another one. If we change the name of a workspace, we must re-edit *dtwmrc*, log out, and log back in again. That process is just too inconvenient, so there's got to be a better way. Time to hit the `man` pages again! After searching for every instance of *workspace*, we find that the workspace has a resource named Title, and you can specify it like this:

```
Dtwm*screen*workspace*title
```

The example given is

```
Dtwm*0*ws1*title
```

After a little while, the answer dawned on me. The workspaces are named ws0, ws1, ws2, etc., and the fancy strings on the buttons are their titles. For some reason, CDE lets you refer to the workspace by its name *or* its title. Therefore, we'll change our F1 through F4 key bindings to use the names ws0 through ws3, and we're finished.

Debugging tips

If only the process were as straightforward as described. It actually took a bit more effort, and we learned a couple of things about debugging configuration changes along the way.

The first tip is this: After you change your *dtwmrc* file, log out, then log in, you should check to see whether you have a *.dt/errorlog* file. If you do, read it. The *.dt/errorlog* file contains diagnostic messages describing any problems that `dtwm` encounters during the current (or last) session.

As an example, when we were trying to make this procedure work, we decided on the keyboard bindings a day before we acutally edited and tested the file. While we were testing the file, we forgot that we had guessed the key names. When the [Alt][PageUp] key combination didn't work, we didn't immediately suspect the name of the key. Instead, we checked the line against the [Alt][F1] line to see if we mistyped anything. Later, when we finally noticed that the *.dt/errorlog* file existed, it gave us the following information:

```
Tue Nov 04 03:40:32 1997
Workspace Manager: Invalid key specification on
➥line 143 of configuration file /home/marco/
➥.dt/dtwmrc
```

Sure enough, our key binding for the [PageUp] key is on line 143.

When you're caught up in the process of solving a problem by experimentation and reading `man` pages, it's easy to forget to log out and log in again. As a result, you may incorrectly assume that a change you made doesn't work because `dtwm` didn't read the configuration file. So be careful, and if something doesn't work the way you expected, just log out, log in, and try the process one more time. ❖

Configuring sar for your system

Before you can tune a system properly, you must decide which system characteristics are important, and which ones are less so. Once you decide your priorities, you then need to find a way to measure the system performance according to those priorities.

In fact, the system activity reporter program suite is a good measuring tool for many aspects of system performance. In this article, we'll introduce you to the `sar` utility, which can give you detailed performance information about your system.

What does sar measure?

Since system tuning involves the art of finding acceptable compromises, you need the ability see the impact of your changes on multiple subsystems. System activity reporter (SAR) programs collect system-performance information in distinct groups. Table A shows how `sar` groups the performance information. The first column shows the switch you give to `sar` in order to request that particular information group. The second column briefly describes the information group.

Table A

Switch	Performance Monitoring Group
A	All monitoring groups
a	File access statistics
b	Buffer activity
c	System call activity
d	Block device activity
g	Paging out activity
k	Kernel memory allocation
m	Message and semaphores
p	Paging in activity
q	CPU run queue statistics
r	Unused memory and disk pages
u	CPU usage statistics (default)
v	Report status of system tables
w	System swapping and switching
y	TTY device activity

One way you can run `sar` is to specify a sampling interval and the number of times you want it to run. So, if you want to check the file-access statistics every 20 seconds for the next five minutes, you'd run `sar` like this:

```
$ sar -a 20 15

SunOS Devo 5.5.1 Generic_103641-08 i86pc
11/05/97

01:06:02  iget/s namei/s dirbk/s
01:06:22     270     397     278
01:06:42     602     785     685
01:07:02     194     238     215
    *    *    *
Average      394     519     438
```

Configuring sar to collect data

Notice that you can't run `sar` right now. If you just try to run the `sar` command without first configuring it, it gives you an error message like this:

```
$ sar -a 20 15
sar: can't open /var/adm/sa/sa03
No such file or directory
```

Sure enough, if you look at the */var/adm/sa* directory, you won't see any files in it—much less that *sa03* file it's complaining about. If you create a blank file, using `touch`, for example, `sar` will start to work. However, why must you do something so strange to make `sar` work? And if you try to run `sar` tomorrow, you'll get a similar error, but this time it will complain about a different file, such as *sa04*.

It turns out that the `sar` program is only one part of the performance monitoring package. Three commands in the */usr/lib/sa* directory also contribute to the whole. The `sadc` command collects system data and stores it to a binary file, suitable for `sar` to use. The shell script `sa1` is a wrapper for `sadc`, suitable for use in `cron` jobs, so it can be run automatically. The `sa2` script is a wrapper for `sar` that forces it to print a report in ASCII format from the binary information in the files `sadc` creates.

If you run the `sa1` script as intended, it then creates a binary file containing all the performance statistics for the day. This file allows `sar` to read the data and report on it without forcing you to wait and collect it. Since you may want to investigate the data a bit later or compare one days' worth of information

against another, the sar, sa1, and sa2 programs name the data file using the same format: */var/adm/sa/saX*, where *X* is the day number. Therefore, when you run sar, one of the first things it does is look for today's binary file. When it doesn't find the file, it prints the error.

The best way to run sa1 and sa2 is from a cron job. Sun provides an example of how to create the cron job instead of forcing you to figure it out for yourself. Thus, if you edit the crontab for the account sys, then you'll see commented-out sample cron schedules for sa1 and sa2, as shown in Figure A.

The first cron schedule uses sa1 to take a snapshot of system performance at the beginning of every hour every day. The second cron schedule adds a snapshot at 20 minutes (:20) and 40 minutes (:40) after the hour between 8:00 a.m. and 5:00 p.m., every Monday through Friday. As a result, you get more detail during business hours and less during evenings and weekends.

The final line schedules sa2 to run at 6:05 p.m. every Monday through Friday to create an ASCII report from the data collected by sa1. This ASCII data is stored using a similar filename convention: */var/adm/sa/sarX*, again where *X* is the day number.

The simplest way to configure sar to run is to edit the sys account's crontab and remove the # signs from the start of the sa1 and sa2 command lines. However, you may want to customize the cron schedules to suit your own preferences. For example, your company might run multiple shifts, and you may want more detailed data. Thus, you can modify the cron job to run sa1 at 15-minute intervals, every business day.

You can't just log into the sys account and edit the cron job, though, because the sys account is usually locked. Instead, you must log in as root, then su to the sys account, like so:

```
$ su
Password:
# su sys
#
```

At this point, be sure to set the EDITOR environment variable to your favorite editor, and edit the crontab file, like this:

```
# EDITOR=vi
# export EDITOR
# crontab -e
```

Now, your favorite editor (vi, in this case) comes up, and you can edit the cron schedules. For our example, we just want to run sa1 every 15 minutes every day, and the sa2 program should generate ASCII versions of the data just before midnight. So we'll change the cron schedule to look like this:

```
0,15,30,45 * * * 0-6 /usr/lib/sa/sa1
55 23 * * 0-6 /usr/lib/sa/sa2 -A
```

Next, we save the file and exit, and crontab will start the appropriate cron jobs for us. That's all you do to configure sar. Once you do so, you can use sar without worrying about the file-open errors any more.

Using the binary data files

Once the system is creating the binary data files, you can use sar without specifying the

Figure A

```
#ident  "@(#)sys        1.5     92/07/14 SMI"   /* SVr4.0 1.2   */
#
# The sys crontab should be used to do performance collection. See cron
# and performance manual pages for details on startup.
#
#0 * * * 0-6 /usr/lib/sa/sa1
#20,40 8-17 * * 1-5 /usr/lib/sa/sa1
#6:05 * * 1-5 /usr/lib/sa/sa2 -s 8:00 -e 18:01 -i 1200 -A
```

The sys account already has prototype entries for running sa1 and sa2, which you can uncomment and use.

Figure B

```
$ sar -up

SunOS Devo 5.5.1 Generic_103641-08 i86pc 11/04/97

00:00:01    %usr    %sys    %wio   %idle
00:15:00       0       0       0      99
00:30:00       0       0       0      99
00:45:00       0       1       0      99
    •       •       •
22:15:00       0       0       0      99
22:30:00       0       0       0      99
22:45:00       1       1       3      95
Average        3       1       4      92

00:00:01  atch/s  pgin/s ppgin/s  pflt/s  vflt/s slock/s
00:15:00    0.00    0.02    0.03    1.82    2.93    0.00
00:30:00    0.00    0.00    0.00    4.35    6.15    0.00
00:45:00    0.00    0.02    0.02   38.95   44.79    0.00
    •       •       •
22:15:00    0.00    0.00    0.00    0.53    0.39    0.00
22:30:00    0.00    0.14    0.19    1.39    1.26    0.00
22:45:00    0.10    2.49    3.34    9.10   14.56    0.00

Average     0.24    2.95    8.32   17.01   18.27    0.00
```

The sar -up command reports detailed information about the CPU and paging use up to the current time.

interval between samples and the number of samples you want to take. Simply specify the data sets you want to see, and `sar` will print all that's accumulated thus far for the day. Therefore, if you're interested in CPU use and paging activity, you'd run `sar` as shown in Figure B on page 9. Since we ran `sar` near the end of the day, and we're sampling every 15 minutes, we're inundated with details. And that's the major problem with detail—it's easy to get swamped.

Getting the bigger picture

While getting a detailed picture of your system is wonderful, you probably don't need or want such a detailed report very often. After all, your job is to manage the system, not micromanage it. Do you think that your company's president monitors the details of the day-to-day operations of the company? Of course not! The president is happy to see the weekly reports showing that the business is chugging along smoothly. It's only when the business is having problems that the president starts to examine and analyze details.

Your role as system administrator is similar to that of the company president: As long as the system is running smoothly, you merely want to glance at a report to see that everything is going nicely. You don't want to delve into a morass of details unless something's awry. Consequently, what we usually want from `sar` isn't a detailed report on all the system statistics, but rather a simple summary.

The `sar` command provides three command-line switches to let you control how you want `sar` to summarize its data. The `-s` and `-e` options allow you to select the starting and ending times of the report, and the `-i` option allows you to specify the reporting interval. So you can see an hourly summary of CPU usage during working hours by using `sar` like this:

```
$ sar -s 08 -e 18 -i 3600 -u

SunOS Devo 5.5.1 Generic_103641-08 i86pc 11/03/97

08:00:00    %usr    %sys    %wio   %idle
09:00:01       0       1       2      97
10:00:00       3       3       1      94
11:00:00       0       0       0     100
12:00:00       0       0       0     100
13:00:00       0       0       0     100
14:00:00       0       0       0     100
15:00:00       5      56      30       8
16:00:01       3      68      24       5
17:00:00       0      11      10      79
18:00:00       0       0       0     100

Average        1      14       7      78
```

If we had a performance problem during the day, we could quickly tell when it occurred using this summary report. Then, we'd adjust our `s`, `e`, and `i` options to focus on the details we're actually interested in seeing. Instead of wading through pages of data, we can be selective.

Conclusion

Once you get `sar` configured, it can capture all the performance statistics for your machine. It's a good idea to browse through the `man` page for `sar` a few times to acquaint yourself with the values it can capture. You needn't understand all of it, especially at the beginning. To start with, it's a good policy to become familiar with the numbers when your system is operating normally. Then, you'll be able to pinpoint which system characteristics are degrading and begin addressing the problems. ❖

SYSTEM-PERFORMANCE TIP

Don't let your directories get too large

If you tend to create directories with many items in them, you may be throwing away performance without even knowing it. This may happen frequently when you write programs to automatically process data for you.

Keep your directories small

Whenever you, or one of your programs, accesses a file, Solaris must first locate it. In order to do so, Solaris must find each component of the path, in order to know exactly where on the disk drive your file is located. For example, if you specify the file */share/work/sort.c*, Solaris starts at the root directory to find the entry for the *share* directory. Next, Solaris reads from the *share* directory looking for the *work* directory; from there, it searches for the file *sort.c*.

Suppose for a moment that Solaris must read the disk for each directory access. If a directory is small, the directory may take only

one or two sectors of disk space. However, when you search through a large directory that consumes more space, you have to read more directory entries from the disk and spend more time looking for the entry you want.

These disk reads are very expensive. If you're trying to read a tiny file, Solaris might spend more time looking for the file than it would reading the file for you.

To make file access as speedy as possible, Solaris maintains two levels of buffering. First, frequently used disk sectors are cached in memory, because once you use a file, chances are good that you'll need to access it again soon. The second level of buffering is the directory-name lookup cache. This is simply a buffer that contains information about the starting location of frequently used directories.

When you access a directory that's not in the buffers, normally Solaris goes through the process we've described to locate the directories. In doing so, it buffers the disk sectors it read to accomplish the task and creates an entry for the directories it searched through.

The problem with large directories is that when Solaris searches through them for a specified entry, it may have to read multiple sectors from the hard drive. Since the disk sector buffer is a limited resource, it will overflow, and Solaris will begin discarding sectors to make room for the new ones.

This, then, is the major problem with large directories. The larger the directory, the more information gets discarded, slowing down all processes as Solaris is forced to re-read sectors from the disk drives. Even worse, when a directory becomes *really* large, the directory itself might overflow the buffer. Thus, even accessing the same directory may force Solaris to re-read data from the disk.

What's the limit?

So, how many files can you place in a subdirectory before performance penalties accrue? We can't give you a hard-and-fast answer. First, only you can determine what performance tradeoffs you're willing to make. Second, directory entries aren't of a fixed size. They vary primarily on the length of the filename. For example, a directory containing five files named *a*, *b*, *c*, *d*, and *e* could take less space than a directory containing a single file with a name such as *GNU_gcc_v2.7.2_Pentium_Optimized_Solaris_2.5.1_i386.pkg.tar.gz*.

Here, a quick check reveals that the current directory contains 155 files, while *sys* contains 400.

```
$ for J in `ls -as1F !grep /$ !awk '{print $2}'`
> do echo "`ls $J !wc -l` $J"
> done
     155 ./
      40 ../
       5 arpa/
     •  •  •
      50 rpcsvc/
       6 security/
     400 sys/
       4 tnf/
      15 vm/
      18 xfn/
```

The *sys* directory averages 25 directory entries per block, and since a block is 512 bytes, the directory entries average 25 bytes. Remember, though, the length of each directory name is proportional to the length of the filename, so directory entries with longer filenames will take more space.

Conclusion

It may not matter if a directory is large if you use it infrequently. However, the directories that you use often should be kept small to

Quick Tip: The best way to determine the size of your directory is to use the `ls -as` command. If you have plenty of normal files intermingled with some of your directories, you can filter the output with `grep`, telling `ls` to display only one column (`-1`) and to add a `/` to the end of directory names (`-F`). Then you can pipe the result to `grep`, telling it to display only lines ending with a `/`, like this:

```
$ cd /usr/include
$ ls -as1F ¦ grep /$
   6 ./
   2 ../
   2 arpa/
   •  •  •
   2 rpcsvc/
   2 security/
  16 sys/
   2 tnf/
   2 vm/
   2 xfn/
```

Listing the */usr/include* directory, as we did here, shows that the */usr/include* directory itself (i.e., the `./` entry) consumes six blocks, and the other large directory, */usr/include/sys*, consumes 16 blocks.

keep the system running at its peak. Typically, you can arrange such a directory as a hierarchical structure, breaking it into several smaller directories and distributing the files among them in a structured fashion.

For example, if you have many users, but only a few log in at any given time, you may want to divide your user-account directory into multiple pieces. Thus, if you typically place user accounts on */acct*, you might break up the directory into */acct/a*, */acct/b*, */acct/c*..., and use the first letter of the account to select the directory in which you'll put the user account.

Remember, each sector of directory information that Solaris reads in forces other data in memory to be discarded. Keeping your directories smaller can help you keep your data in RAM and give you the highest possible performance. ❖

QUICK TRICK

Compressing your files to save disk space

As we mentioned last month in the article "Using find to Locate Unneeded Files," you never seem to have enough space on your disk drives. While they're relatively inexpensive, there's no point in wasting space. You may want to compress your little-used files so they'll consume less disk space.

Compressing your files

File compression is done all the time on the Internet. It saves disk space on the servers, *and* it decreases the time it takes to transfer the file. You've probably already downloaded, uncompressed, and used files in this way.

You can use several programs to compress your files. Two that come with Solaris are `pack` and `compress`. The `pack` command is older and doesn't compress a file as tightly as `compress` does, so now it's rarely used. (In fact, the `man` page for `pack` claims that text files are reduced to 60-75 percent of their original size, while `compress`' `man` page claims reductions to 50-60 percent.) When you compress a file with `pack`, it appends a *.z* to the end of the file so you can tell that it's a `pack` file.

Sun's standard file-compression program is the `compress` program. When Sun supplies patches, drivers, etc., they're often compressed with `compress`. You can easily recognize a file compressed with `compress` because it appends a *.Z* to the end of the filename.

Compatibility: a heavy burden

One problem with `compress` is that it's written to be compatible with other versions of UNIX, as well as different revisions of the same UNIX. Because of this, `compress` hasn't been updated to take advantage of the latest compression technologies.

The GNU project, which creates wonderful free software, created its own file-compression

Quick Tip: If someone renames the file and doesn't preserve the extension, you may not recognize whether or not the file is compressed, or even what program was used. This frequently occurs when you transfer files between Solaris and MS-DOS or Windows computers. Often the case of the extension is changed, so it's easy to mistake a compressed file for a packed one, and vice versa. Typically, you'll notice this situation when you try to decompress the file, in which case the program will complain that the file is in the wrong format, with these lines:

```
$ uncompress B
B.Z: not in compressed format
```

In any case, if you can't determine what type of file you're dealing with, you can use this file command to decipher the file types:

```
$ file*
A:      USTAR tar archive
B.z:    packed data
C.Z:    compressed data block compressed 16
bits
D.gz:   gzip compressed data - deflate method ,
 original file name
```

utility, called `gzip`. Since `gzip` is newer, it takes advantage of more recent compression algorithms. You can recognize a `gzip` file because it will add a *.gz* to the end of the file after it's compressed. (Please note: If the filename ends with *.tar*, and it's too long to add *.gz* to the end, `gzip` will replace the *.tar* extension with *.tgz*.)

Another benefit `gzip` offers is that it contains a tag that specifies the algorithm used to compress the file. This way, rather than having to write a brand-new program when someone creates a new compression algorithm, you can just add a new tag and the appropriate code to `gzip` to support it.

Because `gzip` can support multiple compression algorithms, it examines the file you're going to compress to decide which compression algorithm works best on your data. Thus, you'll get better compression if you use `gzip` to compress your files.

For example, we compressed a file with `pack`, `compress`, and `gzip` using the following code:

```
$ cp A B; pack B
pack: B: 21.5% Compression
$ cp A C; compress C
$ cp A D; gzip D
$ ls -s1
total 32448
12160 A
9552 B.z
6528 C.Z
4208 D.gz
```

After fiddling around with a calculator for a few moments, we found that `compress` squeezed out 46 percent, and `gzip` squeezed out 65 percent of the file, saving 4MB. (We saved more than 8,000 blocks of 512 bytes.)

You can get `gzip`, in either source code or binary form, from many places on the Internet. The *sunsite.queensu.ca/sun/solaris_2.5.html* site provides it in `pkgadd` format for SPARC machines, while for x86-based machines, the equivalent page is *sunsite.queensu.ca/sun/solaris_2.5_x86.html*.

Compressing a directory

Frequently, you'll find that files you no longer need are grouped together in directories. Then, you want to compress all the files in a directory. However, none of the tools we've discussed works directly on a directory, though `gzip` comes close since it allows you to recursively traverse a directory structure and compress all the files found there.

If you're stuck with using `compress` or `pack` to compress the files, you can effectively do the same by using the `find` command to locate the files, then pass them (via `xargs`) to the compression program. When you do, be sure to specify the `-type f` option to ensure that you only try to compress files, like this:

```
$ find /opt/SUNWddk -type f -print |
> xargs compress
$ du -s JS
42161   JS
```

The good side of this directory compression technique is that it's quick and simple. You also have all the files in their original locations, so you can easily locate the file you want and uncompress it.

If you're going to archive the file to tape, you'll probably want to archive the directory first, then compress the archive file. Afterwards, you can remove the original directory. For example:

```
# tar cf JS_tar JS
# ls -s JS_tar
69048 JS_tar
# gzip JS_tar
# ls -s JS_tar.gz
28936 JS_tar.gz
# rm -rf JS
```

You'll notice that the uncompressed archive file (69,048 blocks) is slightly smaller

Quick Tip: You might choose not to compress a directory if you won't gain much for your efforts, so you need to know in advance the total size of the directory. You can use the `du -s` command to find the size of all the files measured in 512-byte blocks. You can print out the space in 1KB blocks by adding the `-k` option:

```
$ du -s JS
69081   JS
$ du -sk JS
34540   JS
```

The *JS* directory holds 34.5MB, so it's a worthwhile candidate for compression. With `gzip`, you can use the `-r` argument to compress all the files in the *JS* directory and subdirectories like this:

```
$ gzip -r JS
$ du -s JS
28961   JS
```

than the directory (69,081). This is because `tar` packs the files closely together in a single file and has no inodes to worry about. On a file system, each file takes up space in fixed-size blocks (e.g., 512 bytes, 1KB), so the last block of each file usually has a little space left over. In a `tar` archive file, the next file may begin immediately after the preceding one ends, so there's no wasted space.

The compressed archive file is a single file, suitable for writing to tape. If you want to restore the directory, you just reverse the steps: Use `gunzip` to expand the file, then use `tar` to rebuild the directory structure.

```
# gunzip js_tar
# tar xf js_tar
```

Now you can work with the newly restored files in your directory.

Quick Tip: Many programs use file extension to indicate the file type, such as *.z* for packed, *.Z* for compressed. The `tar` command, however, gets the filename from you, the user. Since a constant extension is such a useful convention, you may want to end a `tar` file's name with *.tar* or *_tar*, as we did in our example.

Closing notes

Compressing little-used files is a relatively cheap way to make large quantities of disk space available for other tasks. If you already use `pack` or `compress` to compress your files, then you may want to investigate `gzip` as an alternate tool. If you *do* switch to `gzip`, be sure you recompress your files. Just decompressing those files and recompressing with `gzip` may save you hundreds of megabytes. ❖

SMALL NETWORK TIP

A simple way to synchronize the computers on your network

Many networks run every day with administrators unconcerned about time synchronization on the individual computers in the network. However, some applications require synchronized computers.

In software development, for example, the `make` command uses the time and date stamp on the source code files to decide which files must be compiled and which may be skipped. If your computers lose synchronization, some files may not be compiled when needed, and others may compile repeatedly because the time stamp shows that the source file is still newer than the object file.

In our shop, we had exactly this problem. Changes we knew had been made wouldn't make it into the latest test release. Similarly, some compile cycles would take far too long. We finally tracked this problem down to its source: the drifting clocks between computers.

Centralized time management

Once we tracked down the problem, we set the date on all the computers and went off to develop software. A couple of months later, the problems began again. This time, we decided to completely eliminate the problem.

Our first attempt was to create a `cron` job on the time server that uses the remote shell (`rsh`) command to execute the `date` command on each computer we wanted to synchronize, like this:

```
rsh Zeus date 1005
```

However, this technique has a couple of minor problems. First, we may not have an account on each of those computers, so we must specify the account on which we want to log in. Second, only the root account may change the time on the computer, and we want to keep the root account secure with a password. Since we have a password on the root account, we can't just tell `rsh` to execute the `date` command as root on the remote computer, using the command

```
rsh Zeus -l root date $TIME
```

To solve the first problem, we created a specific user account, named *time*, to run the `date` command. We then made the `date` command always run as if it were the root user. To

do so, we turned on the suid permission bit—as well as the execute permissions for the `date` command—on each computer, like this:

```
# chmod 4711 /usr/bin/date
```

Making these concessions gives us the TimeSync script, as shown in Listing A.

Listing A: *TimeSync script*

```
#! /bin/ksh

# Get the current time on this machine, the
# 'Time Server'.
TIME=`date '+%m%d%H%M'`

# Set the time on all the client machines
# using the time account.
rsh Zeus -l time date $TIME
rsh Hydra -l time date $TIME
rsh Cerebus -l time date $TIME
rsh Loki -l time date $TIME
rsh Thor -l time date $TIME
```

Now we have a centrally managed method to synchronize time on the network. However, a user can still set the time and date on an individual computer, potentially wreaking havoc. Also, since we made the `date` command run as root, we opened a possible security loophole.

We could easily fix these problems by making a new copy of the `date` command and placing it in a secure area that only the time account could access. Instead, we decided to make each user responsible for synchronizing his or her own computer to the time server as the user may desire.

Distributed synchronization

Placing the responsibility for computer synchronization on the user turned out to be a better solution for us. This way, we could publish a method that anyone wanting synchronization could use, and we relied on the software developers to administer themselves in this regard.

The method we used to do this was very similar to the one we just described. But rather than making the server use `rsh` to execute the `date` command on all the clients, we make each client ask the date from the server, like this:

```
rsh Mercury date
```

Again, we don't want to put every possible user account on our time server, so we created the time account itself on it. Now, everyone can use the same account to find the current "correct" time like so:

```
$rsh Mercury -l time date
 Mon Nov2 12:35:54 EST 1997
```

At this point, we only need to make the local computer set the time to the value just stated by Mercury, our time server—a simple enough process. If we pass the proper format string to the `date` command on Mercury, then we can pass the results directly into the `date` command on the client machine.

We opted to use the full precision offered by Solaris. Thus, when we install a computer on the network, we needn't worry about setting the date or time at all. It may be set automatically to the correct date, from the century to the second. Our statement that gets the current time from Mercury looks like this:

```
rsh Mercury -l time /usr/xpg4/bin/date
+'%m%d%H%M%C%y.%S'
```

Please note that we're using the XPG4 version of the `date` command to get the century (`%C`) from date. If you didn't install the SUNWxcu4 package, you don't have this version of the `date` command. In that case, go ahead and use the standard `date` command, and use 19 instead of `%C` in the format string. (Don't forget to change it in two years!)

First, we create a script file, named `Sync`. This script reads the time from the time server and sets the local time, as shown in Listing B.

Listing B: *Sync script*

```
#! /bin/ksh

# Ensure that we're using the XPG4 version of date
# (it supports the %C format specifier).
DATE=/usr/xpg4/bin/date

# Specify the host and account that will provide
# the 'correct' date and time.
TSRV="Mercury -lTime"
TIME=`rsh $TSRV $DATE +'%m%d%H%M%C%y.%S`
date $TIME
```

We're almost finished. All that's left is to periodically run the `Sync` script to synchronize the computer to our reference computer. Again, only the root account may set the date and time on a machine. In this case, however, there's no problem. Since the root account already runs several `cron` jobs, we'll just add another one to periodically set the date and time. So simply

PERIODICALS MAIL

SunSoft Technical Support
(800) 786-7638

Please include account number from label with any correspondence.

add a crontab entry that runs our `Sync` script every hour, as in the following line, and never again be bothered about the time:

```
0 * * * * /usr/local/Sync
```

NTP

These two methods of synchronizing the time on your computers are certainly very simple. However, the synchronization achievable with these techniques is only accurate to within a second or two of your time server. And just how well does your time server keep track of time?

Solaris 2.6 brings plenty of new features to the table. One of these is the NTP protocol for time synchronization of multiple computers on a network. Using NTP, you can synchronize your computers to amazingly tight tolerances, since this protocol even performs some adjustments for network delays on your network. (Please note: You can also get NTP in source code form, so you can install it on any version of Solaris.)

One of my team downloaded a copy of `xntp`, which is a freely available program that implements the NTP protocol for synchronizing time on a network of computers. It's a very powerful tool, but much more than we really need, especially when some of the network computers were relatively underpowered with no RAM to spare for yet another application.

Consequently, we decided to run `xntp` on only one computer, having it keep track of the correct time on one of the available time servers on the Internet. Then, we used this as our time server and used our simple synchronization scheme for all the other computers on our network.

Summary

Many networks don't really need to worry about time synchronization. For many of those that do, tight precision isn't a necessity—a few seconds either way isn't significant. For these applications, one of our synchronization schemes is simple to implement and use. However, those applications that require more precise synchronization should consider NTP, even though it brings more complexity and overhead with its improved precision. ❖

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