

IN THIS ISSUE

1
Housekeeping with cron

3
IPC with Doors on
Solaris 2.6

8
Numerical computing in
Solaris

10
Process distribution and
load balancing—problems
and solutions

14
Solaris Q&A

16
Quick Tip

Visit our Web site at
www.zdjournals.com/sun

INSIDE SOLARIS™

Tips & techniques for users of SunSoft Solaris

Housekeeping with cron

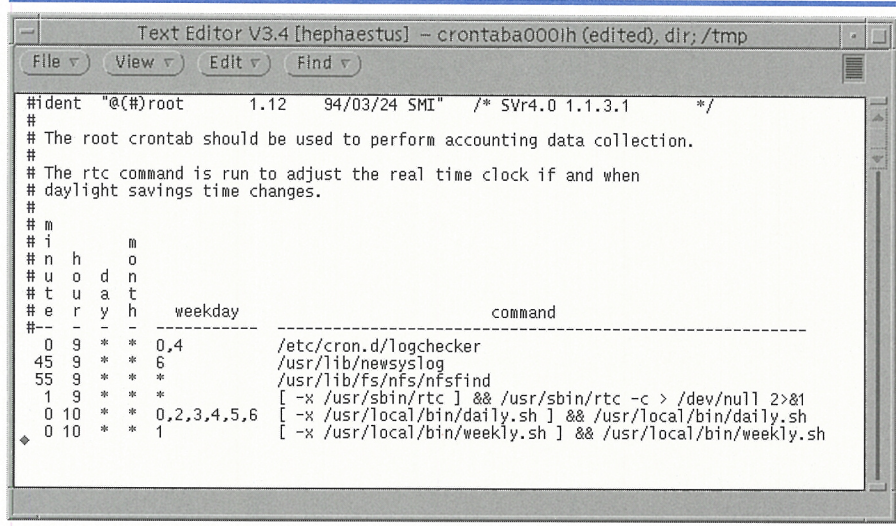
by Don Kuenz

Solaris includes a background job scheduler named cron, which allows you to periodically run housekeeping shell scripts to keep your system in top shape. Housekeeping often includes: backing up file systems, detecting nearly full file systems, removing unwanted files, and trimming logs. In this article, we'll show you how to use cron and show you a couple of useful housekeeping scripts that you can use as a starting point for your own scripts. Although both scripts

in this article use the Bourne shell, you're free to use most any command within cron.

For most of us, the best time to perform housekeeping is outside of normal business hours. It's always a good idea to invest in an Uninterruptible Power Supply (UPS) to protect your computer from unexpected gaps in electrical power. When you use cron, the UPS becomes mandatory. You also need access to root if you plan to use the scripts described in this article.

Figure A



```
Text Editor V3.4 [hephaestus] - crontaba0001h (edited), dir: /tmp
File View Edit Find
#ident "@(#)root 1.12 94/03/24 SMI" /* SVr4.0 1.1.3.1 */
#
# The root crontab should be used to perform accounting data collection.
#
# The rtc command is run to adjust the real time clock if and when
# daylight savings time changes.
#
##
# m
# i m
# n h o
# u o d n
# t u a t
# e r y h weekday command
#-- --
0 9 * * 0,4 /etc/cron.d/logchecker
45 9 * * 6 /usr/lib/newsyslog
55 9 * * * /usr/lib/fs/nfs/nfsfind
1 9 * * * [ -x /usr/sbin/rtc ] && /usr/sbin/rtc -c > /dev/null 2>&1
0 10 * * 0,2,3,4,5,6 [ -x /usr/local/bin/daily.sh ] && /usr/local/bin/daily.sh
* 10 * * 1 [ -x /usr/local/bin/weekly.sh ] && /usr/local/bin/weekly.sh
```

This is the content of our root account crontab.

Using cron

By default, Solaris starts a cron daemon in the `/etc/rc1.d/K70cron` startup script. Normally, cron reads tables found under `/var/spool/cron/crontabs` and logs output to `/var/cron/log`. `/var/spool/cron/crontab` contains one table named after each user who enables and uses cron. A table is little more than a specially formatted text file. **Figure A**, on the cover, shows the content of a typical table.

Each row of a cron table contains six columns. The first five columns specify when a command will launch, while the command itself appears in column six. You must keep in mind your host's time zone when you specify hours and minutes. Our host happens to use Greenwich Mean Time (GMT), but your host may very well use a different time zone.

Both of our housekeeping scripts appear at the end of the root crontab, after Solaris' four default entries. **Figure B** shows a daily script named `/usr/local/bin/daily`, which is scheduled to launch every day except Tuesday at 10:00 GMT (3:00 AM Mountain Time). **Figure C** shows a weekly script named `/usr/local/bin/weekly`, which is scheduled to launch on Tuesday at 10:00 GMT.

Normally, cron mails both standard output and errors from a command to the owner of the crontab. We prefer to only see errors, so we route each script's standard output to `/dev/null`. That way, root only receives an E-mail when something goes wrong.

Although you might be tempted to directly edit cron tables with an editor, Solaris expects you to use a command named `crontab` to change cron tables. The partial syntax of crontab follows:

Figure B

```
Text Editor V3.5.1 - daily.sh (edited), dir: /home/don/incoming/croncobb
File View Edit Find
#!/bin/sh
#
# Daily housekeeping script.
#
PATH=$PATH:/usr/local/bin
export PATH
echo `date`
#
# Set these variables to the number of days you'll allow unused files to hang
# around.
#
maxLastAccess=+0
maxLastModification=+0
#
# Set this variable to rm command (handy for script debugging).
#
rmCommand="/bin/lis -ld"
rmCommand="/bin/rm -f"
#
# Remove core dumps, unnamed binaries, objects, and temporary files with an
# access time greater than maxLastAccess days.
#
find /export/home -xdev '(' -name core -o -name a.out -o -name '*.o' \
-o -name '#*' -o -name '*.#*' -o -name '*.CKP' -o -name '*.nfs*' ')' \
-atime $maxLastAccess -exec $rmCommand {} ';'
#
# Remove backup vi files with a modification time greater than
# maxLastModification days.
#
find /var/preserve/* -mtime $maxLastModification -exec $rmCommand {} ';'
#
# Remove left-over, Samba print files (WinXX host names are: dionysus, cyclops,
# charybdis, and ares).
#
find /tmp/* '(' -name 'dionys.*' -o -name 'cyclop.*' -o -name 'charyb.*' \
-o -name 'ares.*' -o -name 'lpg.*' ')' \
-atime $maxLastAccess -exec $rmCommand {} ';'
#
# Remove directories under /tmp with access times greater than maxLastAccess.
#
find /tmp/* '(' ! -name . ! -name lost+found ')' -type d \
-mtime $maxLastAccess -exec $rmCommand {} ';'
#
#
# Report on any file systems, which are at least 90% full.
#
df -k | awk 'NF == 6 && $5 >= "90%" {printf "File system %s is %s
full.\n", $1, $5}'
#
# Backup file systems, which are most likely to change on a daily basis.
#
echo "Backing up hephaestus"
cd /
mt -f /dev/rmt/1 erase
umount /export/home
ufsdump 0cuvf /dev/rmt/1 /dev/rdisk/c0d0s4
mount /export/home
mt -f /dev/rmt/1 offline
```

This is the content of our `/usr/local/bin/daily.sh`.

Figure C

```
Text Editor V3.5.1 - weekly.sh (edited), dir: /home/don/incoming/croncobb
File View Edit Find
#!/bin/sh
#
# Weekly housekeeping script.
#
PATH=$PATH:/usr/local/bin
export PATH
#
# How many logs do you want to keep?
#
maxLogs=5
#
# Set this variable to rm command (handy for script debugging).
#
rmCommand="/bin/lis -ld"
rmCommand="/bin/rm -f"
#
# A function to archive and remove logs
#
archiveLog()
{
    # $1 = path & file name of log
    dname=`dirname $1`
    bname=`basename $1`
    fname=$1
    fnameDate=$fname.`date +%Y%m%d.%H%M`
    # remove link, keeping oldfile with date
    rm -f $fname
    # create new link
    touch $fname
    ln $fname $fnameDate
    # only keep the newest maxLogs files, remove older file(s)
    for oldfname in `ls -lt $fname.*`
    do
        if [ `echo $1|wc -c` -gt $maxLogs ]
        then $rmCommand $oldfname
        fi
    done
    echo $ix
}
#
# call archiveLog once for each log you wish to archive
#
archiveLog /var/adm/aculog
archiveLog /var/adm/messages
archiveLog /var/adm/sulog
archiveLog /var/adm/vold.log
archiveLog /var/adm/wtmp
archiveLog /var/adm/log/asppp.log
archiveLog /var/log/syslog
archiveLog /var/cron/log
archiveLog /var/saf/_log
archiveLog /var/saf/zsmom/log
#
# Backup file systems, which are most likely to change on a weekly basis.
#
cd /
ufsdump 0uvf /dev/rmt/0n /dev/rdisk/c0d0s4
ufsdump 0uvf /dev/rmt/0n /dev/rdisk/c0d0s7
ufsdump 0uvf /dev/rmt/0n /dev/rdisk/c0d0s12
ufsdump 0uvf /dev/rmt/0n /dev/rdisk/c0d0s11
mt -f /dev/rmt/0 offline
```

This is the content of our `/usr/local/bin/weekly.sh`.

```
crontab [-e] [-l]
```

The `-e` option tells `crontab` to edit your table, while the `-l` option tells it to list your table. When you use the `-e` option, `crontab` defaults to using the rather cryptic `ed` program as an editor. A better solution is to tell it to use your favorite editor by setting an environmental variable named `EDITOR` to the name of your favorite editor *before* you invoke `crontab`:

```
EDITOR=/usr/openwin/bin/textedit
```

```
export EDITOR
```

```
crontab -e
```

If you forget to export an environment variable named `EDITOR` beforehand, just remember to type the letter `q` at your first input opportunity. At that point, `q` causes `ed` to quit without saving.

A daily script

Our daily housekeeping script removes unwanted files, lists file systems with less than 10 percent free space, and backs up files that are most likely to change on a daily system. On our host, the `/home` file system is most likely to change each day. Although other file systems will also change during a week, we're willing to expose the other file systems to potential data loss based on the notion that it's easy to find and re-install recently acquired data. For newly acquired, precious

data, we manually perform an immediate backup outside of cron.


Unwanted files accumulate in a variety of ways. The kernel creates a core dump whenever something unexpected happens. Samba can create temporary print files that hang around until a reboot. Programmers tend to create object files, which they may forget to remove. Our daily script looks for and removes all of these types of files.

A weekly script

Our weekly housekeeping script trims log files and backs up all file systems that are likely to change. This includes every file system except `/`, `/var`, and `/usr`.

Log files can grow until they completely fill your file system. The first part of `weekly.sh` keeps log sizes down by trimming them. When `weekly.sh` trims a log, it creates an empty file using the default name of the log file with the date and time suffix following. Then, it creates a link with the default name pointing to the current log.

Summary

Solaris hosts need periodic housekeeping to stay in top shape. Cron is a great tool for starting periodic jobs outside of normal business hours. 

Don Kuenz is a computer consultant. You can contact him at gtcs.com/assoc/ks/don.

Solaris API features

IPC with Doors on Solaris 2.6

by Abdur Chowdhury

Doors is a fast IPC (InterProcess Communication) call application programming interface (API) used in Solaris 2.5 and 2.6. The Doors API has been documented for general use since Solaris 2.6. The Doors concept originated from the "Lightweight Remote Procedure Call"¹ in the late 1980's. Sun's research operating system, Spring, then used those research ideas. The Spring operat-

ing system used Doors as a method of storing object state and interfaces between domains. Spring was the testing bed for many of Sun's new additions to its commercial operating system. Today, Doors is an important topic of discussion because it's the fastest IPC mechanism available for Solaris. In this article, we'll discuss the concept of Doors on Solaris and the Doors API: A simple client/server.

The Doors concept

The concept of Doors is very simple. When the client process makes a request from the server using `door_call(3x)`, the Doors library creates a shuttle. The shuttle contains information for the server on the client `pid`, `group`, `signals`, and `scheduling group`. The shuttle² puts the client process in the sleep state and the server in the run state. The server wakes and creates a new thread to handle the client's request. The server processes the information, returns the shuttle, destroys the thread, and puts the server in the sleep state and the client in the run state.

Why is this IPC mechanism better than other IPCs? The above procedure is conceptually the same as a protecting shared memory, an RCP call, or even a pipe. The advantages of Doors are found in the kernel. The Solaris 2.6 kernel has the ability to run the client's thread in the servers' process space without an expensive context switch. The Solaris kernel can make optimizations for zero I/O copies between the server process and the client process space by simply mapping pages of memory to both processes.

We can use a synchronization object called a *shuttle* to transfer information from the client space to the server space—like signaling information and `procfs` operations—we can use a synchronization object called a shuttle. The shuttle is responsible for marking the current thread as sleeping and marking the server thread as running. Then, it passes control directly to the server thread.

Table A lists Sun's published timings comparing Doors to other IPC mechanisms. The timings are for SPARCstation 10 (dual processor, 40 MHz SuperSPARCTM processors).

Table A: IPC Timings from Sun[2]

IPC Mechanism	usecs
Doors	66
SVR4 Semaphores	142
Pipes	175
SVR4 Messages	270
ONC-RPC	1020

The Doors API

The Doors API consists of eight Door calls and uses two related calls, `attach(3)` and `detach(3)`. We'll give you a brief overview of the API and develop a simple client and server application to demonstrate use of the Doors API.

First, the user-level library, *libdoor.so*, implements the Door API. Any process can become a Door server. We can create a Door interface by using the `door_create` function. Now, this call returns a Door descriptor that's similar to a file descriptor. Next, the kernel provides the descriptor as the return value of `door_create`. In addition, the kernel keeps track of the process and Door descriptor pairs to ensure that a process can't fabricate a false Door descriptor. The `door_create` call simply sets up the call back function or the exported function entry point. Then, the API mandates a specific argument set for the Door function that you write.

Although this may seem restrictive, the API really provides all the needed constructs to pass data—including Door descriptors. The ability to pass Door descriptors from process A to process B gives this API a tremendous amount of flexibility. You can pass file descriptors from one process to another with Doors.

Now, the serving process must export the interface for other processes to use. The concept of creating yet another naming service isn't a good software practice, so the process must export its interface as part of the file system, much like named pipes or fifos are used. A process uses the `attach` system call to attach a Streams-based Door descriptor to a file system name space object.

Next, a server creates a Door export via the file space with `attach` and waits for clients to request services. The Door API handles the actual creation of server threads. This paradigm provides simple client/server architecture with thread support.

The client process only needs to open the file created by the server process. The client will receive a Door descriptor instead of a file descriptor, and the client simply calls `door_call` with the new Door descriptor. Then, the Door library takes over for the inter-process communication.

All `door_calls` have the same calling arguments, a Door descriptor and the address of a `door_arg_t` structure. The `door_arg_t` structure contains the passing data, the size of the data, any Door descriptors that need to be sent, and the memory buffer for the returning results. Because all calls to `door_call` are the same, no Interface Declaration Language (IDL) is needed to define the interface between the client and server processes. It's the responsibility of both the client and server to agree on the information format.

The server calls the `door_return` function when it has completed the client's request. The arguments for `door_return` are a subset of the `door_call` function. The server process returns the data with the `door_return` call.

Remaining Door calls

Following is a brief description on the remaining Door calls. Servers may use `door_revoke` to revoke access to a Door descriptor. If any requests are being currently handled, that request will finish, but any additional calls will not be honored. We can use the `door_info` call to get information on a door-like the process ID of the server. The server process uses the `door_cred` call to get the user ID, group ID, and process ID of the calling process. We can use the `door_server_create` call when the default thread creation needs to be

modified by the server. The man page gives a very good example on the server creating new threads for the client with a modified stack size. The calling process uses the `door_bind` call to bind to a particular door. We can use the `door_bind` call when a private pool of threads is created. The `door_unbind` call is the antecedent of `door_bind`. This API is beyond the scope of this article, but it's worth exploring when developing ideas for actual Door servers.

Doors example

In [Listing A](#), we show you two programs. The first is a simple server that opens a file descriptor and returns it to the client making the request. The second is a client that makes a request from the server for a particular file. The given makefile compiles the example codes.

The server example is a little longer than necessary for the basic implementation. This demonstrates the need for cleanup when exiting. First, the server creates a thread to handle all signals. When the server process receives a signal to terminate, the server must close the stream from the file system before exiting. The stream must be closed so that the entry point can be reused when the server is restarted.

The server implementation is simple. The server calls `door_create` with the `DoWork`

Listing A: Client and server using Doors

Listing: Makefile

```
CC = gcc
LIBS = -lpthread -ldoor
PROGS= s c

all: $(PROGS)

client.o: client.c
    $(CC) -c client.c

server.o: server.c
    $(CC) -c server.c

c: client.o
    $(CC) -o c client.o $(LIBS)

s: server.o
    $(CC) -o s server.o $(LIBS)

clean:
    rm -rf *.o s c
```

Listing: Mydoor.h

```
#ifndef MYDOOR
#define MYDOOR

#ifdef __cplusplus
extern "C" {
#endif

#define DDSTREAM "DoWork"

#ifdef __cplusplus
}
#endif

#endif /* MYDOOR */
```

Listing: Client.c

```
#include <stdio.h>
#include <string.h>
#include <door.h>
#include <pthread.h>
#include <thread.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/time.h>

#include "mydoor.h"

static const char* ddstream = DDSTREAM ;

int
main(int argc, char **argv)
{
    /* Vars */
    int dd = 0 ; /* Doors descriptor */
    int ret_val = 0 ; /* return values from
functions */
    door_arg_t d_args ;
    char results[1024] ;
    int fd ;
    char message [ 128 ] ;

    strcpy ( message, "Inside Solaris!\n" ) ;

    dd = open ( ddstream, O_RDONLY ) ;
    if ( dd < 0 )
    {
        fprintf ( stderr, "Error in opening a door-
server descriptor: %d\n",
            errno ) ;
        exit ( -1 ) ;
    }

    d_args.data_ptr = NULL ;
    d_args.data_size = 0 ;
    d_args.desc_ptr = NULL ;
    d_args.desc_num = 0 ;
    d_args.rbuf = results ;
    d_args.rsize = sizeof(results) ;

    ret_val = door_call(dd, &d_args);
    if ( ret_val != 0 )
    {
        fprintf ( stderr, "Error in
door_call to door descriptor: %d\n",
            errno ) ;
        exit ( -1 ) ;
    }
}
```

```
fd = d_args.desc_ptr-
>d_data.d_desc.d_descriptor ;
printf("desc_ptr=%x, desc_num=%d fd=%d\n",
    d_args.desc_ptr, d_args.desc_num, fd) ;

write(fd, message, strlen(message) ) ;
close ( fd ) ;

/* close the door */
close ( dd ) ;
return (0);
}
```

Listing: Server.c

```
#include <stdio.h>
#include <door.h>
#include <pthread.h>
#include <thread.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include "mydoor.h"
static const char* ddstream = DDSTREAM ;

/* Close Door on File System */
void
CloseTheDoor (int sig)
{
    fdetach ( ddstream ) ;
    exit ( -sig ) ;
}

/* Signal handler */
void *
sighandle (void *arg)
{
    struct sigaction sigact ;
    arg = NULL ;

    /* With signals we can handle we will clean up */

    /* SIGINT */
    sigact.sa_flags = 0 ;
    sigact.sa_handler = CloseTheDoor ;
    sigemptyset ( &sigact.sa_mask ) ;
    sigaddset(&sigact.sa_mask, SIGINT) ;
    sigaction ( SIGINT, &sigact, NULL ) ;

    /* SIGTERM */
    sigact.sa_flags = 0 ;
    sigact.sa_handler = CloseTheDoor ;
    sigemptyset ( &sigact.sa_mask ) ;
}
```

Listing: *Server.c (continued)*

```
sigaddset(&sigact.sa_mask, SIGTERM) ;
sigaction ( SIGTERM, &sigact, NULL ) ;

/* SIGQUIT */
sigact.sa_flags = 0 ;
sigact.sa_handler = CloseTheDoor ;
sigemptyset ( &sigact.sa_mask ) ;
sigaddset(&sigact.sa_mask, SIGQUIT) ;
sigaction ( SIGQUIT, &sigact, NULL ) ;

sigignore(SIGPIPE) ;
sigignore(SIGHUP) ;
sigignore(SIGABRT) ;

for(;;)
{
    sleep( 10000 ) ;
}

return ( (void *) NULL ) ;
}

/* Work function provided by the server */
void *
DoWork (void *cookie, char *argp,
        size_t arg_size, door_desc_t *dp, size_t n_desc)
{
    int fd ;
    door_desc_t darray[1] ;
    int n ;

    fd = open("server.dat", O_RDWR | O_CREAT | O_TRUNC,
0600);

    printf ( "In DoWork: getting fd for client.\n" ) ;

    darray[0].d_attributes = DOOR_DESCRIPTOR ;
    darray[0].d_data.d_desc.d_descriptor = fd ;

    door_return (NULL,0,darray,1) ;
    return ( (void *) NULL ) ;
}

int
main(int argc, char **argv)
{
    /* Vars */
    int dd = 0 ; /* Doors descriptor */
    FILE *fd ;
    int ret_val = 0 ; /* return values from functions
*/
```

```
sigset_t      set ;
pthread_attr_t detached_attr ;

/* make signal handler thread detached */
pthread_attr_init ( &detached_attr ) ;
pthread_attr_setdetachstate( &detached_attr,
PTHREAD_CREATE_DETACHED ) ;

do
{
    errno = 0 ;
    pthread_create(NULL, &detached_attr,
sighandle, NULL) ;
} while ( errno == EAGAIN ) ;

/*
Block all signals in the main thread. Any other
threads created
by the main thread will also block all signals
*/
sigfillset(&set) ;
pthread_sigmask(SIG_SETMASK, &set, NULL) ;

/* Make sure we have a file to use as a
interface to the server */
if ( (fd = fopen ( ddstream, "w+" )) == NULL )
{
    fprintf ( stderr, "File creation for doors
interface failed: %d\n",
            errno ) ;
    exit ( -1 ) ;
}
else
{
    fclose ( fd ) ;
}

dd = door_create ((void*)DoWork, NULL, 0) ;

ret_val = fattach ( dd, ddstream ) ;
if ( ret_val != 0 )
{
    fprintf ( stderr, "Error in fattach to door
descriptor: %d\n",
            errno ) ;
    exit ( -1 ) ;
}

/* Wait forever */
while ( 1 ) pause ( ) ;

return (0);
}
```


function pointer and then attaches the Door descriptor to the file system and waits forever.

Also, the client's implementation is just as simple. The client opens the server's exported file entry point, uses that descriptor from the open system call as the Door descriptor, and calls `door_call`. When the call is finished, the client continues with the result in the pre-allocated buffer.

Conclusion

In this article, we provide a brief overview on how Doors works, a brief description of the API, and an example using it for a simple server and client. In conclusion, the Doors API is the fastest IPC available by Sun on Solaris. Doors makes it simple to develop multithreaded servers. This API is very flexible and is worth further exploration when designing fast servers under Solaris. The Doors concept is full of promise and is why Sun is one of the leaders in new and innovative ideas, but the API must be expanded to work over the network to be truly useful for many situations.

At the time of writing this article, the API has been ported to Linux, so other

UNIX implementation may soon follow. At this time, the Door IPC mechanism doesn't work across the network, but this is being addressed by Sun for future releases. Also, we give a special thanks to Andy Spitzer for his help. 

1 - B. Bershad, T. Anderson, E.Lazowska and H.Levy, "Lightweight Remote Procedure Call", *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 134-154.

2 - Sun Developer Benchmarks Fall 1996
solaris.javasoft.com/developer/news/devnews/fall96/doors.html

Abdur Chowdhury is the manager for the Systems Engineering and Integration Lab at the University of Maryland at College Park. He's also working on his Ph.D. in computer science on distributed systems. Abdur received his BS in 1994 and MS in 1996 in Computer Science at George Mason University. He has authored many papers on process migration, fault tolerant routing protocols, and information retrieval topics. You can reach Abdur at abdur@isr.umd.edu.

NUMERICAL COMPUTING

Numerical computing in Solaris

by Paul A. Watters

Have you ever written a numerical simulation using C that produces widely variable results when one or two parameters have been slightly tweaked? Does your database produce historical reports based on a C module that greatly overestimates the profitability of your clients' business interests? Many of us have made simple errors in numerical computing with C in the past, particularly those involved in floating-point computations. Unfortunately, these errors can sometimes adversely affect the desired outcomes for applications, with extra debugging efforts that often inflate programming costs.

Know thy enemy

The success of numerically-intensive programs written by computing professionals in both science and industry can be reasonably assured if strategies are developed *a priori* to deal with anticipated and well-documented numerical errors. These strategies must often go beyond what is recommended in many programming texts (e.g., explicit typecasting of all computations involving mixed data types). In this article, we'll outline several components that you can incorporate into an overall strategy for numerical computing when using the Solaris C compiler. This

facilitates the development of robust numerical applications by implementing IEEE Standard 754 for Binary Floating Point Arithmetic.

Making numerical programming easy

The implementation of this standard makes numerical programming easier and ensures that our software meets specific quality criteria. For example, special undefined numerical quantities, such as infinity (Inf) and Not a Number (NaN) are available for handling run-time errors through the compiler. In the past, this facility has generally only been available in interpreted numerical languages (e.g., Matlab). In addition, standard exception handlers are provided for the classic numerical programming errors of underflow and overflow. This is particularly important for complex simulations, such as neural network modeling, where a single undetected overflow error passed to a non-linear function could render the results of the entire simulation useless. Custom handlers can also be included to enhance those already provided by default under the IEEE standard.

A plan of attack

How can we ensure that floating-point errors don't ruin our programs? There are three main issues that you must address in an overall strategy. Furthermore, each issue must be thought through carefully before you embark on any project that involves numerical computation. The first issue involves number representation as a hardware problem, and understanding the effects that it may have on our programs. Although number representation is clearly different for machines with individual operating systems and hardware, the flexibility of Solaris being implemented on several hardware platforms can sometimes lead to confusion about the physical representation of numbers in each individual hardware platform. For example, the SPARC architecture has a high-endian representation of floating point numbers, while the INTEL platform has a low-endian representation. Although these differences might not appear to be very important at first glance (particularly for small-domain computations), if data is shared between computers of different architectures but the same operating system in binary format (e.g., shared database

files), you must be careful to correctly translate individual number representations. Failing to do so might result in an apparent (but incorrect) increase in monthly profits computed from such shared database records, for example, which could have dramatic consequences for our clients.

Once the representation of numbers is understood, the second issue that you must address is how errors can arise as a result of the binary representation of (generally) base-10 floating-point numbers. Many programmers explicitly declare levels of floating-point precision (e.g., single or double precision, and an extended double in Solaris, which occupies four 32-bit words), and feel re-assured that their compiler would automatically take care of overflow and/or underflow errors. This is not, however, always the case. [Listing A](#) on page 10 shows how easily an overflow error can occur in even an integer-based program. This program doesn't have any of the safeguards that should be taken for specifying integer precision, etc. The following is the output showing the higher exponents of two computed using a simple formula:

```
x^1=2
x^2=4
x^3=16
x^4=128
x^5=2048
x^6=65536
x^7=4194304
x^8=536870912
x^9=0
x^10=0
```

As you can see, the higher-order exponents are incorrectly computed as zero, which would have an obvious impact on our applications' results. Fortunately, Solaris has some built-in methods for dealing with these kinds of errors (including a capacity to trap division-by-zero errors, for example, which might normally result in a run-time core-dump). This is usually achieved by defining a user-specified signal handler using `sigfpe(3)`.

After dealing with possible errors in numerical representation and processing, the third issue in using Solaris for numerical computing involves optimizing our programming to either minimize time or CPU utilization, depending on local priorities. Optimization can be invoked on the

command line by flags such as `-fast` and `-xO[1,2,3,4,5]`. Many of these options can take advantage of SPARC and INTEL architecture by using register variables for temporary variables, loop unrolling, dead-code elimination, etc. Users can also improve their own coding by proper loop nesting and inlining function calls (which can reduce execution time, but increase the size of the executable). Another method of optimization involves using the most efficient implementation of a particular algorithm. In fact, many of these are available in the Solaris development environment, which has three main math libraries. These are *libsunmath* (math library), *libmvec* (vectorized math library) and *libcopt* (optimized math library). Our use of these libraries or of those provided with Numerical Recipes, for example,

Listing A: C source code demonstrating overflow error

```
#include <stdio.h>

main()
{
    int i=0, j=0, x=2, pow=2;


    for (i=0; i<10; i++)
    {
        for (j=1; j<=i; j++) pow *= x;
        printf("x^d=%d\n", i+1, pow);
    }
}
```

will complement any of the automated optimizations that a particular compiler might make on our behalf.

Conclusion

C is often thought of as the last language of choice for numerical programming because early compilers failed to meet the same standards that scientists and engineers, for example, had come to expect from formula-based languages such as FORTRAN. However, the new Solaris compilers allow us to develop intensely numerical programs using a compiled, functional programming language with more certainty that our results are accurate, or at the very least, to place limits of the interpretability of our results.

Further reading

The bible of numerical computing is *Numerical Recipes*, which is now freely available on-line at nr.harvard.edu/nr/nronline.html. This book contains generic code for C, FORTRAN, BASIC, and Pascal, providing function libraries for computing almost any possible mathematical function in an efficient manner. 

Paul A. Watters is a research officer in the Department of Computing at Macquarie University, Australia. He has developed a number of numerically-intensive simulations (e.g., neural networks) using the Solaris development environment.

Distributed Computing

Process distribution and load balancing—problems and solutions

by H-W Schlote

The computing power of many workstations can be combined to reach the area of super-computing. But these benefits can be accompanied by many problems. In this article, we'll explore some possible solutions.

In the past, super-computers were used to solve complex computing tasks. But, things have changed. Prices of middle-ranged workstations have fallen to very affordable amounts, while their computing power has increased nearly reaching the area of super-

computing. Therefore, today many industries are substituting former super-computers with UNIX workstation clusters. Famous films such as *Toy Story* (computed on a Sun workstation farm) and *Titanic* (Digital UNIX DEC Alpha and Linux-PCs) used UNIX clusters. Another example is the RSA-contest, where thousands of computers worldwide searched for crypt-keys.

We must classify distributed computing into several fields. One field is known more commonly by the name *massive parallel computing*. This means that one computer with dozens or even hundreds of processors use shared memory. Communication between several distributed jobs can be performed using toolkits like parallel virtual machine (PVM), or the message passing interface (MPI) both provided by GNU.

In this article, the term *distributed computing* refers to a software product that accepts requests and starts the corresponding job on one of many hosts in a LAN. The prerequisite for using distributed computing is the possibility to divide the computing task into several sub-tasks.

A case study

The following example will show both the power of using many workstations together and the potential pitfalls. Suppose you need to process geographical data of a country. The conventional (super-computer) approach would be to solve the problem in one step using the hierarchical memory structure of a super-computer.

Now, consider dividing the country into parts. The amount of data per computing task (a subtask of the main goal) is thereby greatly reduced, and the subtask can be solved by a UNIX workstation. With a load balancing and process distribution system, the several subtasks are distributed over the available workstations. This way, costs, time, and effort are significantly reduced.

Problem: security hole

Most load balancing and process distribution systems must be run with superuser privileges. They communicate over the network, sometimes not only in the LAN but in a WAN. So, you have one more program running with superuser privileges listening on ports open to hackers all over the world—just another security hole. We

know of only one system capable of running with normal user privileges—PDS. We'll give you a list of available process distribution software at the end of this article.

Problem: uncertainty

Imagine it's Friday afternoon. There's a final deadline on Monday for delivering processed data. All your programming work is done and this morning you solved the last (known) error. The processing takes about two days on the 30 workstations available. So, you start your jobs and go home with a really good feeling. Unfortunately, there is some kernel panic or power failure (the cleaning personnel accidentally unplugged the power cord) or anything else on one of the machines working for you. To make it worse, this machine had almost finished a job on which lots of others depend.

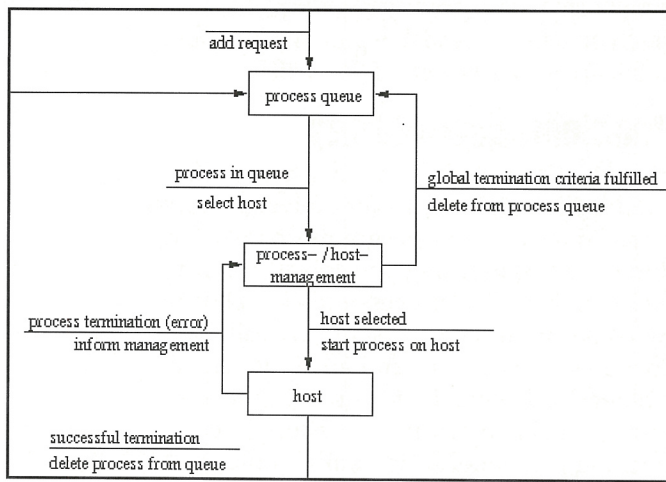
So, your whole process gets sent back to the beginning and you have to explain to your boss what happened and why you used the only machine that failed. You also have to explain why this job was running there, why you didn't take some other machine, why you didn't look for the jobs on Saturday and Sunday etc.

There's a possibility for you to use some high availability (HA) attempt to avoid such a problem. But you would have to build your network with every machine being redundant, which leads to unnecessarily high costs. What you want are central machines like database servers being built redundant (for example, a big Sun enterprise server with redundant backplane, processors, memory, interfaces, etc.) or mirrored with an HA attempt like Sun's SPARC-cluster-HA and SPARC-cluster-PDB. All workstations should be managed by a process distribution system. This needs to be done not only to keep up with the work (which is easy) if anything goes wrong, but to also start a failed process anew if it's important to you and you think that the failure lies within the hardware (network, computer hardware, or operating system) and not within your software. [Figure A](#) on page 12 shows a graphical representation of a process distribution system with fault management.

Conflicts between users and jobs

Often you use the same workstations as workplace computers and to compute distributed

Figure A



This is a process distribution system with fault management.

jobs. This may lead to problems if the distributed jobs hinder the user. There are multiple ways to avoid this kind of conflict. A good process distribution system should offer all possibilities as options.

First, you can generally define that no processes should be run during office hours. This is the most effective method, but it wastes a lot of computing time. The typical Sun machine has more than one processor and can perform more than one job at a time. So why shouldn't a job run on this machine if sufficient memory is installed and sufficient computing power exists?

The second way to avoid conflicts between users and distributed jobs is to set the maximum number of jobs to be run on a machine explicitly less than the number of CPUs online. Outside of office hours, the system can be configured to use all CPUs.

The third approach would be to use resources more generally. That's to say, the job uses n CPUs, m MB RAM, d MB disk space etc. The process distribution system should try to avoid starting more jobs on a machine than resources that are available.

Only the first (very strict) approach, wasting computing time, will guarantee that conflicts won't occur. The success of the other possibilities depends on how well people work together and how good the communication is between the different users.

In our opinion, a combined approach of the second and third way is the best. Specifi-

ying the resources that will be needed can help you avoid trashing a machine by starting jobs on it that need more memory than is actually installed. Also, reducing the number of jobs to be run on workplace machines during office hours will help you produce a friendly working atmosphere.

Equal rights

Consider that Paul is one user of a workstation cluster and is starting a huge number of parallel jobs. A few minutes later Mary wants to start just one small job. How should the distribution system perform in this case? First-come-first-serve wouldn't be well accepted by the users. The next time, Mary would start her job directly, which could possibly confuse the distribution system. There must be some tricky algorithm implemented into a good process distribution system. This algorithm must assure that Mary's job would be started immediately after Paul's first job finished.

Priorities

The process distribution system should provide some kind of priority scheduling. There are always more and less important things to calculate. But using several queues will lead to new problems. You need some kind of aging algorithm. A job started with minor priority must be executed sometime. If there are always jobs with higher priorities, the minor priority job would never be executed, which isn't what we want. There must be some algorithm increasing the priority of a job with time passing.

If the process distribution system uses different queues for high, medium, and low priorities, you'd find a job with low priority in the high priority queue after some time. This is confusing.

Another approach is to use one queue with priorities and have the priorities of older jobs eventually get higher. This method is straightforward and produces clear results.

Which directory to choose?

Where will a distributed job be executed? Or more precisely: in which directory? The UNIX-command `rsh` executes the job given in the home directory of the user. But, if compilation jobs need to be distributed, they must

be started in the current directory—which must be mounted via NFS, of course.

In a heterogeneous UNIX cluster, however, even the location of home directories may be different. And, on each machine there may be some local directory with sufficient space available for distributed jobs that aren't named the same on two machines.

There should be a way to tell the process distribution system in which directory to change for execution. Also, you should be able to configure the system so it knows how the local directories are named, providing there's sufficient free disk space on each machine.

Different classes

There may be some software available only for workstations of a specific vendor's UNIX version. Also, there may be some job which needs a large amount of computing power.

Many situations require dividing available computers and jobs to be computed into classes. Good software should provide an easy way to do this.

Listing A: Shell script wrapper for heterogeneous distribution and platform dependent execution of programs

```
#!/usr/bin/ksh

ARCH='pvmgetarch'

if [ ! -d $HOME/bin/$ARCH ]; then
  echo "Architecture $ARCH not supported"
  echo ""
  exit 1
fi

ARCH_DIR=$HOME/bin/$ARCH

#####
# ... finally start the application ...

cmd='basename $0'

case $cmd in
  foo ) exec $ARCH_DIR/foo $*;;
  * ) echo "$cmd may not be"
      echo "started from here";;
esac
```

Heterogeneous clusters

Process distribution in heterogeneous UNIX clusters has one more problem: A Sun-SPARC executable doesn't run on a DEC-Alpha or an SGI workstation. Therefore, you must write a wrapper shell script using the pvmgetarch-script (included in the PVM distribution mentioned earlier) to find out on which platform the job has started. **Listing A** shows a simple example for such a shell script wrapper.

Available software

Digital Equipment developed load balancing over 15 years ago for VAX/VMS. Nowadays, almost every hardware vendor has his own clustering software, which is sometimes combined with high availability approaches. These solutions, however, are always platform-specific.


There is some process distribution software for heterogeneous UNIX clusters. Some examples that are in the public domain are DQS/NQS, LSF (Platform Computing, Canada), and PDS (SUFFIX, Germany).

DQS/NQS works well if you don't need reliability in case of network instabilities or other problems. LSF, however, keeps working if problems occur. Yet, PDS is the only product that provides the ability to automatically restart failed jobs (contact pds@suffix.de). Besides, PDS doesn't need superuser privileges, and therefore isn't just another security hole in your network.

On the other side, you can expand and configure LSF using shell scripts. For example, if you want to take the round-trip-time (RTT) between two hosts into account, you can specify it. The disadvantage of this approach lies with the different implementations in the two main streams of today's UNIX systems. A System V ping returns "host is alive" per default. The BSD equivalent permanently outputs the RTT to the given host. In order to obtain LSF, contact Platform Computing, Canada, or the local sales office (i.e., Science & Computing in Germany).

All available systems offer solutions to avoid conflicts between users and jobs and between several users (see the "Equal Rights" heading). All approaches dealing with priorities are very different. DQS/NQS doesn't have an aging algorithm (at least none that

we know of) and doesn't handle queues with different priorities. LSF has several queues with several priorities. And, PDS has a tricky priority-scheduling algorithm, including aging using one queue.

Also, LSF starts the distributed job in the current working directory. PDS starts a job either in the current working directory or in a directory configured within the process distribution system, depending on the user's choice (given via command line argument). 

H-W Schlote was born in 1969 in Soltau, Germany. After studying physics at TU Braunschweig, he worked for about one year for a software firm in Braunschweig. In 1996, H-W (known better as Harvey) founded his own firm, SUFFIX. Today SUFFIX is a group of three UNIX specialists, mainly operating in air guidance and car navigation projects. He can be reached at H.Schlote@suffix.de.

Solaris Q & A:

Ping the Solaris Dude

by Robert Owen Thomas

Does something about Solaris have you puzzled? Ping the Solaris Dude today! Submit questions to robt@cymru.com.

How to use `ndd`

One of the most powerful, yet least used and understood commands in Solaris is `/usr/sbin/ndd(1M)`. The tool `ndd(1M)` modifies the kernel parameters of the drivers that use TCP/IP. Such drivers as `/dev/tcp`, `/dev/ip`, and `/dev/hme` can be tuned through the use of the `ndd(1M)` command.

One of the *first* and *most important* things to realize when using `ndd(1M)` is that you're modifying kernel parameters. This can be very dangerous business if you aren't careful, because kernel parameters can affect everything and everyone using your system. Remember this kernel tuner's mantra: Don't tune that which you don't fully understand. Now, having delivered this warning, the good news is that anything you change with `ndd` is only in effect until the next reboot. So, if something should go awry, you can reboot to recover.

You can use the `ndd(1M)` tool as both a query tool and a set tool. For example, if you want to know if your dual-homed host is acting as a gateway, simply type

```
orc # ndd /dev/ip ip_forwarding
1
```

So what does this output mean? Generally speaking, a `1` means "on" and a `0`

means "off". In this case, you enable IP forwarding between the two networks. Now, let's check this on a single homed host:

```
pudge # ndd /dev/ip ip_forwarding
0
```

Sure enough, IP forwarding is disabled. Next, let's try an example with ARP. Perhaps you're curious about the length of time an entry is maintained in the ARP cache? With the help of `ndd(1M)`, we can find out. We try:

```
pudge # ndd /dev/arp arp_cleanup_interval
300000
```

`ndd(1M)` reports all time values in milliseconds. So, 300,000 milliseconds equates to approximately five minutes.

How do I know what to tune?

Unfortunately, the `ndd(1M)` main page doesn't exhaustively document all of the various tuneables for each driver. However, `ndd(1M)` is in some way self-documenting—all you have to do is ask!

For example, [Listing A](#) shows all of the TCP tuneables. These are all the variables under the `/dev/tcp` device driver. All of the variables with the second column (read and write) are tunable. This doesn't mean, however, that they should be tuned. Remember the kernel tuner's mantra! Now, list out all of the variables for `/dev/arp`, `/dev/ip`, and `/dev/udp`.

Listing A: All off the TCP tunable variables shown with the `ndd` command

```
pudge # ndd /dev/tcp \?  
tcp_conn_req_max (read and write)  
tcp_conn_grace_period (read and write)  
tcp_cwnd_max (read and write)  
tcp_debug (read and write)  
tcp_smallest_nonpriv_port (read and write)  
tcp_ip_abort_cinterval (read and write)  
tcp_ip_abort_interval (read and write)  
tcp_ip_notify_cinterval (read and write)  
tcp_ip_notify_interval (read and write)  
tcp_ip_ttl (read and write)  
[ Output snipped due to volume. ]
```

Don't be worried if you don't know what each of the variables represents. You'll likely never need to tune more than a select few.

What variables should be tuned?

Setting the value of the tunables with `ndd(1M)` is very easy. The `-set` option is all you need. For example, to disable IP forwarding, simply enter:

```
orc # ndd -set /dev/ip ip_forwarding 0
```

Instantly, IP forwarding is disabled. Watch out! Recall that this will revert to the default (enabled) at the next reboot. More on that later.

There are several other variables that we recommend you tune. For example, the now infamous smurf attack relied upon hosts responding to broadcast ICMP ECHO_REQUEST messages. This can easily be disabled by entering the following:

```
orc # ndd -set /dev/ip ip_respond_to_echo_broadcast 0  
And:  
orc # ndd -set /dev/ip ip_respond_to_address_mask_broadcast 0
```

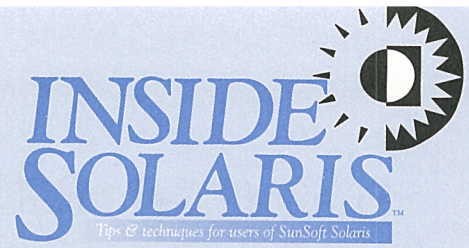
If your host is a gateway, you may also wish to prevent directed broadcasts from passing through your gateway host. This can be accomplished with:

```
orc # ndd -set /dev/ip ip_forward_directed_broadcasts 0
```

Some of the more common questions we see are related to the hme (Fast Ethernet) settings. Yes, `ndd(1M)` can help here as well. For example, if you wanted to force the hme driver to use only 100 Mbit/s full-duplex, you could enter

```
orc # ndd -set /dev/hme adv_100fdx_cap 1  
orc # ndd -set /dev/hme adv_100hdx_cap 0  
orc # ndd -set /dev/hme adv_10fdx_cap 0  
orc # ndd -set /dev/hme adv_10hdx_cap 0  
orc # ndd -set /dev/hme adv_autoneg_cap 0
```

This tells the hme driver to set the speed to 100 Mbit/s, the duplex to full, and disable all other settings. The last entry, `adv_autoneg_cap`, disables the automatic negotiation process.



Inside Solaris (ISSN 1081-3314) is published monthly by
ZD Journals, 500 Canal View Boulevard, Rochester, NY 14623.

Customer Relations

US toll free (800) 223-8720
Outside of the US (716) 240-7301
Customer Relations fax (716) 214-2386

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to

ZD Journals Customer Relations
500 Canal View Boulevard
Rochester, NY 14623

Or contact Customer Relations via Internet E-mail at zdjcr@zd.com.

Editorial

Editor Garrett Suhm
Copy Editors Rachel Krayer
Christy Flanders
Taryn Chase

Contributing Editors Don Kuenz
Abdur Chowdhury
Robert Owen Thomas
Paul A. Watters
H-W Schlote

Print Designer Lance Teitsworth

General Manager Jerry Weissberg
Editor-in-Chief Joan Hill
Editorial Director Michael Stephens
Managing Editor Kent Michels
Circulation Manager Renee Costanza
Print Design Manager Charles V. Buechel
VP of Operations and Fulfillment Michael Springer

You may address tips, special requests, and other correspondence to

The Editor, *Inside Solaris*
500 Canal View Boulevard
Rochester, NY 14623

Editorial Department fax (716) 214-2387

Or contact us via Internet E-mail at sun@zdjournals.com.

Sorry, but due to the volume of mail we receive, we can't always promise a reply, although we do read every letter.

Postmaster

Periodicals postage paid in Louisville, KY.

Postmaster: Send address changes to

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

Copyright

Copyright © 1998, ZD Journals, a division of Ziff-Davis. ZD Journals is a trademark of Ziff-Davis. *Inside Solaris* is an independently produced publication of ZD Journals. All rights reserved. Reproduction in whole or in part in any form or medium without express written permission of Ziff-Davis is prohibited. ZD Journals reserves the right, with respect to submissions, to revise, republish, and authorize its readers to use the tips submitted for personal and commercial use.

Inside Solaris is a trademark of Ziff-Davis Inc. Sun, Sun Microsystems, the Sun logo, SunSoft, the SunSoft logo, Solaris, SunOS, SunInstall, OpenBoot, OpenWindows, DeskSet, ONC, and NFS are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc. Other brand and product names are trademarks or registered trademarks of their respective holders.

Price

Domestic \$99/yr (\$9.00 each)
Outside US \$119/yr (\$11.00 each)

Back Issues

To order back issues, call Customer Relations at (800) 223-8720. Back issues cost \$9.00 each, \$11.00 outside the US. You can pay with MasterCard, VISA, Discover, or American Express.

ZD Journals publishes a full range of journals designed to help you work more efficiently with your software. To subscribe to one or more of these journals, call Customer Relations at (800) 223-8720.

To see a list of our products, visit our Web site at www.zdjournal.com.

SunSoft Technical Support
(800) 786-7638

Please include account number from label with any correspondence.

Making the tuning permanent

Once you've worked out your tuning strategy, and once you've successfully and rigorously tested your changes, it's time to make the changes permanent. This can't be accomplished, however, with **ndd(1M)**.


To make the changes permanent, you must add the lines to `/etc/system`. Nonetheless, the lines added to `/etc/system` are very close to the same arguments you've given to **ndd(1M)**.

Let's take the hme variables as an example. To make the changes permanent, edit `/etc/system` to add

```
# set hme to only use 100 Mbit/s, full duplex
# ROT, 17 June 1998
set hme:hme_adv_100fdx_cap = 1
set hme:hme_adv_100hdx_cap = 0
set hme:hme_adv_10fdx_cap = 0
set hme:hme_adv_10hdx_cap = 0
set hme:hme_adv_autoneg_cap = 0
```

At reboot, the changes you've made with **ndd(1M)** will take effect automatically. Don't forget to add comments to your changes in `/etc/system`.

Conclusion

The **ndd(1M)** command is a powerful tool, and a must for any system administrator's toolbox. However, use it wisely. As with any kernel tuning tool, it has the capability to cause great harm. When you use it wisely, it can greatly enhance the performance and security of your Solaris systems. 

Robert Owen Thomas is an aspiring blues guitarist earning his living as a UNIX and networking consultant. He can be contacted through E-mail at robt@cymru.com, or visit his web site at www.cymru.com/~robt.

Quick Tip

"rm -r" TIPS

"rm -r file..." every administrator in the crowd already knows the rest of this story!

Tip #1

`rm -r` should *always* be preceded with `pwd` to make sure you're in the right directory. Don't rely on your prompt to tell you. The `sh` command may put you in `/"` as it did me!

Tip #2

Always use the full path when running `rm -r` because if you miss the dot ("`.`") as I did (`rm -r ./dir-name`), you'll have to restore an entire file system.

Tip #3

It's not a good idea to `rm -r` sub-directories with the same name as your root-level file systems and directories (see Tip #2).

Any other `rm -r` tips should be E-mailed to: forsythe@tusco.net so that I can quit making these mistakes on my own!

