Using *expect* to Automate System Administration Tasks

Don Libes

National Institute of Standards and Technology
Metrology Bldg, Room A-127
Gaithersburg, MD 20899
libes@cme.nist.gov

ABSTRACT

UNIX system administration often involves programs designed only for interactive use. Many such programs (**passwd**, **su**, etc.) cannot be placed into shell scripts. Some programs (**fsck**, **dump**, etc.) are not specifically interactive, but have poor support for automated use.

expect is a program which can "talk" to interactive programs. A script is used to guide the dialogue. Scripts are written in a high-level language and provide flexibility for arbitrarily complex dialogues. By writing an **expect** script, one can run interactive programs non-interactively.

Shell scripts are incapable of managing these system administration tasks, but **expect** scripts can control them and many others. Tasks requiring a person dedicated to interactively responding to badly written programs, can be automated. In a large environment, the time and aggravation saved is immense.

expect is similar in style to the shell, and can easily be mastered by any system administrator who can program in the shell already. This paper presents real examples of using **expect** to automate system administration tasks such as **passwd** and **fsck**. Also discussed are a number of other system administration tasks that can be automated.

Keywords: **expect**, **fsck**, interaction, **passwd**, password, programmed dialogue, security, shell, Tcl, UNIX, **uucp**

Reprinted from *Proceedings of the Fourth USENIX LISA Large Installation Systems Administration (LISA) Conference*, Colorado Springs, CO, October 17-19, 1990.

January 21, 1992

Introduction

UNIX system administration often involves using programs designed for interactive use. Many such programs (**passwd**, **su**, etc.) cannot be placed into shell scripts. Some programs (**fsck**, **dump**, etc.) are not specifically interactive, but have little support for automated use.

For example the **passwd** command prompts the user for a password. There is no way to supply the password on the command line. If you use **passwd** from a shell script, it will block the script from running while it prompts the user who invoked the shell script.

Because of this, you cannot, for example, reject passwords that are found in the system dictionary, a common security measure. It is ironic that security *was* the reason that **passwd** was designed to read directly from the keyboard to begin with.

passwd is not alone in this recalcitrant behavior. Many other programs do not work well inside of shell scripts and quite a few of these are crucial tools to the system administrator. Examples are **rlogin**, **telnet**, **crypt**, **su**, **dump**, **adb**, and **fsck**. More problems will be mentioned later.

The problem with all of these programs is not the programs themselves, but the shell. For example, the shell cannot see prompts from interactive programs nor can it see error messages. The shell cannot deal with interactive programs this way because it is incapable of creating a two-way connection to a process. This is an inherent limitation of classic UNIX shells such as **sh**, **csh** and **ksh** (from here on generically referred to as simply *the shell*).

expect – An Overview

expect is a program that solves the general problem of automating interactive programs. **expect** communicates with processes by interposing itself between processes (see Figure 1). Pseudo-ttys are used so that processes believe they are talking to a real user. A high-level script enables handling of varied behavior. The script offers job control so that multiple programs can be controlled simultaneously and affect one another. Also, a real user may take and return control from and to the script whenever necessary.

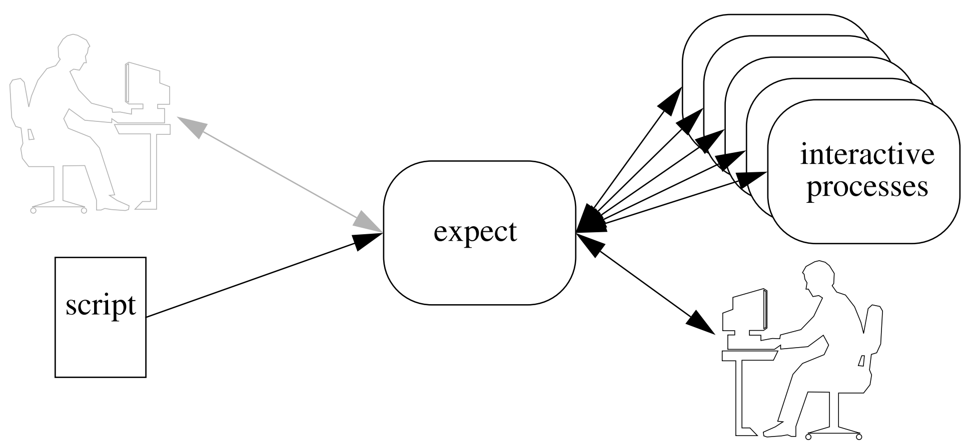

Figure 1. **expect** is communicating with 5 processes simultaneously. The script is in control and has disabled logging to the user. The user only sees what the script says to send and is essentially treated as just another process.

expect is a general-purpose system for solving the interactive program problem, however it solves an unusually large number of problems in the system administration arena. While the *UNIX style* is to build small programs that can be used as building blocks in the construction of other programs using shells and pipelines, few system administration programs behave this way.

Traditionally, little time was spent designing good user interfaces for system administrator tools. The reasons may be any or all of the following:

- System administrators were experienced programmers, and therefore didn't need all the hand-holding that general user programs require.
- Programs such as **fsck** and **crash** were run infrequently, so there was little point spending much time on such rarely used tools.
- System administration tools were used in extreme conditions, considered not worth programming for because of their difficulty or rarity. It was more cost-effective to solve the problem by hand in real-time.
- System administrators solved problems in site-dependent ways, never expecting their underdesigned programs to be propagated widely.

Whatever the reason, the result is that the UNIX system administrator's toolbox is filled with representatives of some of the worst user interfaces ever seen. While only a complete redesign will help all of these problems, **expect** can be used to address a great many of them.

Example – passwd

The **expect** script in Listing 1 takes a password as an argument, and can be run non-interactively such as by a shell script. A shell script could prompt and reject easily guessed passwords. Alternatively, the shell script could call a password generator. Such a combination could create large numbers of accounts at a time without the system administrator having to hand-enter passwords as is currently done.

Admittedly, the script reopens the original security problem that **passwd** was designed to solve. This can be closed in a number of ways. For example, **expect** could generate the passwords itself by directly calling the password generator from within the script.

The scripting language of **expect** is defined completely by Libes [1][2] and Ousterhout [3][4]. In this paper, commands will be described as they are encountered. Rather than giving comprehensive explanations of each command, only enough to understand the examples will be supplied.

set – Sets the first argument to the second (i.e., assignment).

In line 1 of the script, the first argument to **set** is **password**. The second is an expression that is evaluated to return the second argument of the script by using the **index** command. The first argument of **index** is a list, from which it retrieves the element corresponding to the position of the second argument. **argv** refers to the arguments of the script, in the same style as the C language **argv**.

```
set password [index $argv 2]
spawn passwd [index $argv 1]
expect {*password:}
send $password\r
expect {*password:}
send $password\r
expect eof
```

Listing 1 Non-interactive **passwd** script. First argument is username. Second argument is new password.

spawn – Runs an interactive program.

The spawned program is referred to as the *current process*. In this example, **passwd** is spawned and becomes the current process. A username is passed as an argument to **passwd**.

expect – Looks for a pattern in the output of the current process.

The argument defines the pattern. Additional optional arguments provide alternative patterns and actions to execute when a pattern is seen. (An example will be shown later.)

In this example, **expect** looks for the pattern `password`. The asterisk allows it to match other data in the input, and is a useful shortcut to avoid specifying everything in detail. There is no action specified, so the command just waits until the pattern is found before continuing.

send – Sends its arguments to the current process.

The password is sent to the current process. The `\r` indicates a carriage-return. (All the "usual" C conventions are supported.) There are two **send/expect** sequences because **passwd** asks the password to be typed twice as a spelling verification. There is no point to this in a non-interactive **passwd**, but the script has to do this because **passwd** doesn't know better.

The final `expect eof` searches for an end-of-file in the output of **passwd** and demonstrates the use of *keyword patterns*. Another one is **timeout**, used to denote the failure of any pattern to match. Here, **eof** is necessary only because **passwd** is carefully written to check that all of its I/O succeeds, including the final newline produced after the password has been entered a second time.

It is easy to add a call and test of `grep $password /usr/dict/words` to the script to check that a password doesn't appear in the on-line dictionary, however, we will leave the illustration of control structures to the next example.

Example – fsck

Many programs are *ostensibly* non-interactive. This is, they can run in the background but with a very reduced functionality. For example, **fsck** can be run from a shell script only with the **-y** or **-n** options. The manual [5] defines the **-y** option as follows:

> *"Assume a yes response to all questions asked by fsck; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered."*

The **-n** option has a similarly worthless meaning. This kind of interface is inexcusably bad, and yet many programs have the same style. For example, **ftp** has an option that disables interactive prompting so that it can be run from a script, but it provides no way to take alternative action should an error occur.

Using **expect**, you can write a script that allows **fsck** to be run, having questions answered automatically. Listing 2 is a script that can run **fsck** unattended while providing the same flexibility as being run interactively. The script begins by spawning **fsck**.

```
spawn fsck
for {} 1 {} {
      expect  eof                             break \
              {*UNREF\ FILE*CLEAR?\ }         {send y\r} \
              {*BAD\ INODE*FIX?\ }            {send y\r} \
              {*?\ }                          {send n\r}
}
```

Listing 2 Non-interactive **fsck** script.

for – Controls iteration (looping).

The language used by **expect** supports common high-level control structures such as **if/then/else**. In the second line, a **for** loop is used which is structured similarly to the C-language version. The body of the **for** contains one **expect** command.

The following **expect** command demonstrates the ability to look for multiple patterns simultaneously. (The backslashes (\) are used to quote characters – in this case whitespace.) In addition, each pattern can have an accompanying action to execute if the pattern is found. This allows us to prespecify answers for specific questions. When the questions `UNREF FILE...CLEAR?` or `BAD INODE NUMBER...FIX?` appear, the script will automatically answer `y`. If anything else appears, the script will answer `n`.

In general, if all questions are known and answerable in advance, a script can be run in the background. With more complex programs it may be desirable to trap unexpected questions and force a user to interactively evaluate them. Listing 3 is a script does exactly this.

```
spawn fsck
for {} 1 {} {
      expect  eof                              break \
              {*UNREF\ FILE*CLEAR?\ }          {send n\r} \
              {*BLK(S)\ MISSING*SALVAGE?\ }{send y\r} \
              {*?\ }                           {interact +}
}
```

Listing 3 User-friendly **fsck** script.

If the script does not match one of the prespecified answers, the last case ({*?\ }) matches. (The ? is necessary to prevent the script from triggering before the entire question arrives.) The **interact** action passes control from the script to the keyboard (actually **stdin**) so that a human can answer the question.

interact – Pass control from script to user and back.

During **interact**, the user takes control for direct interactions. Control is returned to the script after pressing the optional escape character. In this script, + is chosen as the escape character by passing it as the argument to **interact**.

A real **expect** script for **fsck** would do several other things. For example, **fsck** uses several statically-sized tables. For this reason, **fsck** is limited to the number of errors of one type that can be fixed in a single pass. This may require **fsck** be run several times. While the manual says this, **fsck** doesn't, and few system administrators know **fsck** that intimately. When run from a shell script, this lack of programmability will cause the system to come up all the way with a corrupt file system (if the return code isn't checked) or be unnecessarily rebooted several times (if the return code is checked).

Example – Callback

The script in Listing 4 was written by a user who wanted to dial up the computer, and tell it to call him back. Since he lived out of the local calling area, this would get the computer to pick up his long-distance phone bills for him.

```
spawn tip modem
expect {*connected*}
send ATDT[index $argv 1]\r
set timeout 60
expect {*CONNECT*}
```

Listing 4 Callback script. First argument is phone number.

The first line spawns **tip** which opens a connection to a modem. Next, **expect** waits for **tip** to say it is connected to the modem. The user's phone number, passed as the first argument to the script, is then fetched and added to a command to dial a Hayes-compatible modem. A carriage-return is appended to make it appear as if a user had typed the string, and the modem begins dialing.

The third line assigns 60 to the variable **timeout**. **expect** actually looks at this variable in order to tell it how many seconds to wait before giving up. Eventually the phone rings and the modem answers. **expect** finds what it's looking for and exits. At this point **getty** wakes up, and finding that it has a dialup line with DTR on it, starts **login** which prompts the user to log in.

Since the script was originally written, we have added a few more lines to automate and verify phone numbers based on the uid running it partly for security, but the fragment shown here was used successfully and forms the heart of our current script. Ironically, we recently noticed a 60Kb equivalent to **callback** on Usenet that had no more functionality than a dozen or so lines of **expect**.

Of course, not all scripts are this short. I'm limited to what can be presented here, and these examples really serve just to give you a feel of what **expect** does and how it can be applied. What is important is that **expect** scripts are small and simple for problems that are small and simple. **expect** obviates the need for resorting to C just because of limitations on the part of the shell.

Example – Intelligent ftp

One of our site administrators wanted to spool files in a directory. Later, a second computer would use **ftp** to pick them up and then delete them from the first computer. His first attempt was to use `mget *` followed by `mdelete *`. Unfortunately, this deletes files that arrive in the window between when the **mget** starts and the **mdelete** starts. The script fragment in Listing 5 solved the problem.

```
spawn ftp
. . .
send ls * lsFile\r              ;expect *success*ftp>*
set lsVar [exec cat lsFile]
exec rm lsFile\r
set len [length $lsVar]
for {set i 0} {$i < $len} {set i [expr $i+1]} {
      set file [index $lsVar $i]
      send get $file\r          ;expect *success*ftp>*
      send delete $file\r       ;expect *success*ftp>*
}
```

Listing 5 Fragment of **ftp** spool script.

The script begins by spawning **ftp**. I have omitted several lines that open a connection followed by sending and confirming the user and password information. The next line sends an **ftp** command to store the list of remote files in a local file called **lsFile**. This command is terminated by a semicolon, allowing the response to be verified with an **expect** command on the same line of the script.

exec – Execute a UNIX command.

exec executes a UNIX command and simply waits for it to complete, just as if it were in a shell script. In line four, **cat** returns the list of files, and their names are stored in the variable, **lsVar**. **exec** is used again in the next line, this time to delete the local file, **lsFile**.

The remainder of the script merely iterates through the variable **lsVar**, sending **get** commands followed by **delete** commands for each file found in the earlier **ls**.

Other examples solved

expect addresses a surprisingly large class of system administration problems which before now have either been solved by avoidance or special kludges. At the same time, **expect** does not attempt to subsume functions already handled by other utilities. For example, there is no built-in file transfer capability, because **expect** can just call a program to do that. And while the shell is programmable, it cannot interact with other interactive processes and it cannot solve any of the examples in this paper.

In this section, more examples will be discussed. Because of space limitations, scripts will not be shown, but all of them have been written and are being used.

Regression testing

Testing new releases of interactive software (**tip**, **telnet**, etc.) requires a human to press keys and watch for correct responses. Doing this more than a few times becomes quite tiresome. Naturally, people are much less likely to run thorough regression tests after making small changes that they think *probably* don't affect other parts of a program.

Regression testing can also be useful for your entire installation. You can make a script that tests all your site's local applications, and run it at after each system upgrade or configuration change.

Automating logins

Many programs have a frequently repeated, well-defined set of commands and another set that are not well-defined. For example, a typical **telnet** session always begins with a log in, after which the user can do anything. To automate this, **expect** has the ability to pass control from the script to the user. At any time, the user can return control to the script temporarily to execute sequences of commonly repeated commands.

At my site, **expect** is heavily used to automate the process of logging in through multiple front-ends and communication switches. In fact, the original reason **expect** was written was to create six windows, each of which automatically logged in to another host to run a demo.

The general idea of automating **telnet**, **ftp**, and **tip** is very useful when dealing with hosts that do not support **rlogin** and **rcp**. But the technique is also useful with native UNIX commands like **su**, **login**, or **rlogin**. **expect** scripts can call any of them, sending passwords as appropriate and then continuing actions as desired. While any of these commands can be embedded in a shell script, the shell has no way of taking control over what happens *inside* of these programs. Subsequent commands from the shell script do not get sent to the new context, but are held up until the previous command has completed so that they can be sent to the original context. **expect** has no such problems switching contexts to continue controlling any of these sessions.

telnet – It's not just for breakfast anymore

telnet also functions as an interface to the exciting world of TCP sockets. **telnet** can be used to access non-**telnet** sockets and query other hosts for their date (port 13), time (port 37), list of active users (port 11), user information (port 43), network status (port 15), and all sorts of other goodies that you might only be able to get if you had permission to log in.

For example, our site regularly runs a script that checks (port 25) what version of **sendmail.cf** each of our local hosts is actually using. If we did this by reading files, we would need permission to log in, or remotely mount file systems and read directories and files on several hundred hosts. Using **telnet** is much easier, albeit a little strange.

su, passwd, crypt and other password–eaters

Programs that read and write **/dev/tty** cannot be used from shell scripts without the shell script accessing **/dev/tty**. An earlier example showed how to force **passwd** not to read from **/dev/tty**. With this technique, you can change its input source to **stdin**, a parameter, or even an environment variable.

As another example, suppose you have typed a command that fails because you weren't root. The typical reaction is to type `su` and then reenter the command. Unfortunately, history won't work in this situation as `!-2` will just evoke the error `-1: Event not found.` The problem is that you want to refer to a command that is now in a different shell instantiation, and there is no way to get back to it.

A solution is to pass the failed command as an argument (via `!!`) to an **expect** script that will prompt you for the root password, invoke **su**, and then feed the original failed command to the resulting superuser shell. If the **expect** script executes **interact** as its last action, you will have the original command executed for you (no retyping), plus you will get a new superuser shell. There is no way to do this with **su** except by resorting to temporary files for your history and a lot of retyping.

A more painful example is **newgrp**. Unlike **su**, **newgrp** does not allow additional arguments on the command line to be passed to the new shell. You must interactively enter them after **newgrp** begins executing. In either case, both **su** and **newgrp** are essentially useless in shell scripts.

Security – The good news is ...

Earlier, I mentioned how to build a script that would force users to choose good passwords without rewriting **passwd**. All other alternatives either rewrite the **passwd** program or ask the user to be responsible for choosing a good password.

On the opposite side of the coin, **expect** can be used to test other sites for secure logins (or to break in, I suppose). Trying to login as **root** using, say, all the words in an on-line dictionary, at all the local hosts at a site would be prohibitively expensive for a human to do. **expect** would work at it relentlessly, eventually finding an insecure **root**, or showing the site to be protected by good passwords.

Questions at boot time

While booting, it is useful to validate important system facts (e.g., **date**) before coming up all the way. Of course, if no one is standing in front of the console (e.g., the system booted due to a power failure) the computer should come up anyway. Writing such a script using the shell is painful, primarily because a read-with-timeout is not directly implemented in the shell. In **expect**, all reads timeout. **expect** can prompt and read from the keyboard just as easily as from a process.

Transferring hierarchies with ftp

Anonymous **ftp** is very painful when it comes to directory hierarchies. Since there is no recursive copy command, you must explicitly do **cd**s and **get**s. You can automate this in a shell script, but only if the hierarchy is known in advance. **expect** can execute an **ls** and look at the results so that you can transfer a hierarchy no matter what it looks like or how deep it is. **expect** supports recursive procedures, making this task a short script. My site regularly retrieves large distributions (e.g., Gnu, X) this way.

Assisting adb and other “dumb” programs

Quite often, vendors provide instructions for modifying systems in the form of **adb** instructions, where some instructions may depend on the results of earlier ones (i.e., “*each time _maxusers is incremented, you must add 16 to _nfile*”). **adb** has no special scripting language that supports such interaction, nor does the shell provide this capability. **expect** can perform this interaction, playing the part of the user, by directly looking at the results of operations, just as a user would.

This technique can be applied to any program. In fact, **expect** can act as an intermediary between the user and programs with poorly-written user interfaces. **expect** normally shows the entire dialogue but can be told not to. Then **expect** can prompt the user for commands such as `show _maxusers` instead of **adb**’s native but cryptic `_maxusers/d`. Translations can also be performed in the reverse direction. A short **expect** script could limit the difficulty of system administrators who have no interest in mastering **adb**. In addition, the ability of system administrators to accidentally crash the system by a few errant keystrokes would be dramatically lessened.

Grepping monster log files

A common command sequence involves looking at a log with, say, **grep**, and then interrupting it (with ^C) after the line of interest appears. Unfortunately, **grep** and other programs are limited to the amount of programmability they have. For example, **grep** can not be directed to stop searching after the first match. A short **expect** script can send an interrupt to **grep** after seeing the first line just as if the user were actually at the keyboard.

With programs that generate log files as large as a gigabyte, this is a real problem. Without **expect**, the only solutions are to let **grep** continue running over the whole file, or to dedicate a human to the task of pressing ^C at the right time. **expect** can cut off the process as soon as possible, mailing the results back the system administrator if necessary.

In general, **expect** is useful for sending odd characters to a process that cannot be embedded in a shell script. **expect** can also execute job control commands (**bg**, **fg**, etc.) in order to mediate between processes that were never designed to communicate with each other. Again, this can relieve a human from the tedious task of interactively monitoring programs.

Administering non-UNIX systems

expect is a UNIX program, yet it can be used to administer non-UNIX systems. How is this possible? Running **telnet** (**tip**, **kermit**, etc.) to a non-UNIX host, it can log in and perform **send/expect** sequences on the remote computer. The operating system or environment of the remote computer is completely irrelevant to **expect**, since all of this is isolated to the **expect** script itself.

This is very useful for system administrators that already have a UNIX computer on their desk but are forced by management to administer another computer. ("*You already administer 20 UNIX* *systems. How much more work could it possibly be for you to administer just one more system? Oh, and it runs VMS.*")

Security

Several of the examples presented have prompted for passwords that are different than the usual UNIX style. Normally, UNIX prompts for passwords directly from **/dev/tty**. This has the unfortunate drawback that you cannot redirect **stdin**. We have shown how to get around that by using **expect**.

Of course, doing this reopens a possible security hole. Unprivileged users can detect passwords passed as arguments by using **ps**. If passwords are stored in files, lapses in security can make plaintext passwords evident to people browsing through your files. Publicly-readable backup media are one of the simplest such security lapses.

If you are at all interested in security, I do not recommend storing plaintext passwords in files. The likelihood of such a password being discovered and abused is just too high. Our users store passwords in files, but only for highly restricted accounts, such as for demos or anonymous **ftp**.

The chances of leaking a password through **ps** are lower, and can be lowered further still by using the smallest possible script around the password prompting program. Such a window is extremely small. Nonetheless, secure sites should not take even this chance.

An alternative is to have **expect** interactively prompt for passwords. If you have an **expect** script that is doing a complicated series of **telnet**s, **ftp**s and other things, the scripts can encode everything but the passwords. Upon running such a script, the user will be only be prompted once for a password, and nothing else. Then **expect** will use that password whenever necessary, and complete all the other dialogue from data pre-stored in files.

In summary, **expect** need not weaken security. Used wisely, **expect** can even enhance security. However, you must use common sense when writing scripts.

Comparison to other system administration tools

This section of the paper can be considered controversy or heresy, as you wish. It is somewhat religious in that the arguments can only be resolved by philosophical choice rather than logic. I have kept it down to a very few reasons to give you only the barest feelings for what I consider is important to understand when choosing **expect** over other system administration tools.

As should be obvious, I think there are very few alternatives to using **expect**. Traditionally, the popular choices have been 1) avoidance and 2) C programming. These are now no longer the only choices.

Shell

The shell is incapable of controlling interactive processes in the way that **expect** can. Nonetheless, certain comparisons between **expect** and the shell are inevitable. In particular, **expect** includes a high-level language that is interpreted and bears a strong similarity to the shell and also to C. In that sense, I see little to argue about since **expect** can do shell-like functions. In a previous paper [1], I have suggested the addition of **expect**'s features to the shell. No one wants to learn yet another shell, and there is no reason why these capabilities cannot be added to the shell.

Perl

A more interesting comparison is with Perl, a language claimed (by the author) [6] to embody the best aspects of the shell, C, **awk**, **sed**, and a number of other UNIX tools. Having spent some time programming in Perl, there is no question in my mind that Perl is capable of solving the same tasks that I have described in this paper. Pseudo-tty packages for Perl have been written and **send/expect** utilities could be written also.

Perl is a very powerful language. It is much richer than the language used by **expect** (or any shell for that matter). This has advantages and disadvantages. The most obvious disadvantage is that Perl's overabundance of options and features simply aren't necessary for the tasks that **expect** addresses. Perl's complexity is reflected in its disk space. The computer on my desk, a Sun 3, requires 270K to store Perl and has a significant startup time. **expect**, on the other hand, is 70K with essentially no startup time. There are other reasons that Perl is not widely applied to certain problems, but completing the discussion deserves a paper of its own.

Instead I will summarize by saying that **expect** is appropriate to only a fraction of the system administration problems that Perl solves. This is intentionally so. **expect** was written to solve a very specific problem, and it does that concisely and efficiently. I think that it fits well with the UNIX philosophy of small tools, unlike Perl which demands a significant investment in mastering its complexity. Given the choice, I predict that most system administrators would choose a tool like **expect** that takes very little effort to learn, rather than entering the world of Perl.

Emacs

Emacs is analogous to Perl in many ways, including its flexibility and overabundance of functionality. Similarly, Emacs can be used to solve these same problems. And for much the same reasons as I gave above, Emacs is inappropriate for the class of problems I have suggested in this paper. Indeed, considering that Emacs has been available for over a decade, and I've never heard of anyone using it this way, I'll proffer that Emacs is so inappropriate for these problems, that it is not surprising this usage has never even occurred to anyone.

Conclusion

UNIX shells are incapable of controlling interactive processes. This has been at the root of many difficulties automating system administration tasks. While the UNIX community is gradually providing better designed tools and user interfaces, even more programs are being written with embarrassingly poor user interfaces at the same time. This is understandable because system administrators give more priority to solving a problem so they can go to the next one, than going back to pretty up an old and working solution.

expect is designed to work with programs as they are. Programs need not be changed or redesigned, no matter how poorly written. Understandably, the majority of system administrators are reluctant to modify a program that works and that they have not written themselves. Most prefer writing shell scripts using the classic UNIX tools philosophy.

expect handles these problems, solving them directly and with elegance. **expect** scripts are small and simple for problems that are small and simple. While not all **expect** scripts are small, the scripts scale well. They are comparable in style to shell scripts, being task-oriented, and provide synergy with shell scripts, both because they can call shell scripts and be called by them. Used judiciously, expect is a welcome new tool to the workbench of all UNIX system administrators.

Acknowledgments

This work was jointly funded by the NIST Automated Manufacturing Research Facility (AMRF, project 734-3385) and Scientific and Technical Research Services (STRS, project 734-3106).

The callback script was written by Scott Paisley. Walter Rowe wrote the **sendmail** version-checking script mentioned in the paper. Thanks to Walter Rowe and Ken Manheimer for their helpful comments on this paper. Sue Mulroney was helpful in correcting my badly grammar.

Availability

Since the design and implementation of **expect** was paid for by the U.S. government, it is in the public domain. However, the author and NIST would like credit if this program, documentation or portions of them are used. **expect** may be **ftp**'d as **pub/expect/expect.shar.Z** from **ftp.cme .nist.gov**. **expect** will be mailed to you, if you send the mail message `send pub/expect/ expect.shar.Z` to **library@cme.nist.gov**.

References

[1] Don Libes, "expect: Curing Those Uncontrollable Fits of Interaction", Proceedings of the Summer 1990 USENIX Conference, Anaheim, CA, June 10-15, 1990.

[2] Don Libes, "*The expect User Manual – programmatic dialogue with interactive programs*", to appear as a NIST IR, National Institute of Standards and Technology, November, 1990.

[3] John Ousterhout, "*Tcl: An Embeddable Command Language*", Proceedings of the Winter 1990 USENIX Conference, Washington, D.C., January 22-26, 1990.

[4] John Ousterhout, "*tcl(3) – overview of tool command language facilities*", unpublished manual page, University of California at Berkeley, January 1990.

[5] AT&T, UNIX Programmer's Manual, Section 8.

[6] Larry Wall, "*Perl – Practical Extraction and Report Language*", unpublished manual page, March 1990.

photo

Don Libes
National Institute of Standards and Technology

Don Libes received a B.A. in Mathematics from Rutgers University and an M.S. in Computer Science from the University of Rochester.

Currently at the National Institute of Standards and Technology, Don is engaged in research that will help U.S. industry measure the standard hack. Unfortunately, NIST does not have a very good sense of humor, so he was forced to write his first book “Life With UNIX” through Prentice-Hall.