expect: Scripts for Controlling Interactive Processes

Don Libes National Institute of Standards and Technology
libes@cme.nist.gov

ABSTRACT: Contemporary shells provide minimal control (starting, stopping, etc) over programs, leaving interaction up to users. This means that you cannot run some programs non-interactively, such as **passwd**. Some programs can be run non-interactively but only with a loss of flexibility, such as **fsck**. This is where the tool-building philosophy of UNIX begins to break down. **expect** crosses this line, solving a number of long-standing problems in the UNIX environment.

expect uses Tcl as a language core. In addition, **expect** can use any UNIX program whether or not it is interactive. The result is a classic example of a little language synergistically generating large power when combined with the rest of the UNIX workbench.

Previous papers have described the implementation of **expect** and compared it to other tools. This paper concentrates on the language, primarily by presenting a variety of scripts. Several scripts demonstrate brand-new features of **expect**.

Keywords: expect; interaction; POSIX; programmed dialogue; shell; Tcl; UNIX

Reprint of *Computing Systems* , Vol. 4, No. 2, University of California Press, Berkeley, CA, November 1991.

1. Introduction

fsck, the UNIX file system check program, can be run from a shell script only with the **–y** or **–n** options. The manual [1] 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 is safer, but almost uselessly so. This kind of interface is inexcusably bad, yet many programs have the same style. **ftp**, a file transfer program, 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.

expect is a tool for controlling interactive programs. It solves the **fsck** problem, providing all the interactive functionality non-interactively. **expect** is not specifically designed for **fsck**, and can handle **ftp**'s errors as well.

The problems with **fsck** and **ftp** illustrate a major limitation in the user interface offered by shells such as **sh**, **csh**, and others (which will generically be referred to as *the shell* in the rest of the paper). The shell does not provide a way of reading output and writing input from a program. This means the shell can run **fsck** but only by missing out on some of its useful features. Some programs cannot be run at all. For example, **passwd** cannot be run without a user interactively supplying the input. Similar programs that cannot be automated in a shell script are **telnet**, **crypt**, **su**, **rlogin**, etc. A large number of application programs are written with the same fault of demanding user input.

expect was designed specifically to interact with *interactive* programs. An **expect** programmer can write a script describing the dialogue. Then the **expect** program can run the "interactive" program non-interactively. Writing scripts for interactive programs is as simple as writing scripts for non-interactive programs. **expect** can also be used to automate parts of a dialogue, since control can be passed from the script to the keyboard and vice versa.

2. A brief overview of expect

The implementation and philosophy of **expect** is described at length by Libes [2]. Briefly, scripts are written in an interpreted language. (A library is available for C and C++ programmers but it will not be further discussed in this paper.) Commands are provided to create interactive processes and to read and write their output and input. **expect**[1] is named after the specific command which waits for output from a program.

The language of **expect** is based on Tcl. Tcl is actually a subroutine library, which becomes embedded into an application and provides language services. The resulting language looks very much like a typical shell language. There are commands to set variables (set), control flow (if, for,

---

1. For readability, **times roman bold** is used for display of file or program names, helvetica for keyword or other language elements, and `courier` for literal strings or code fragments.

continue, etc), and perform the usual math and string operations. Of course, UNIX programs can be called (exec). All of these facilities are available to any Tcl application. Tcl is completely described by Ousterhout [3][4].

expect is built on top of Tcl and provides additional commands. The spawn command invokes a UNIX program for interactive use. send sends strings to a process. expect waits for strings from a process. expect supports regular expressions and can wait for multiple strings at the same time, executing a different action for each string. expect also understands exceptional conditions such as timeout and end-of-file.

The expect command is styled after Tcl's case command which matches a string against a number of other strings. (Whenever possible, new commands were modeled after existing Tcl commands so that the language remained a coherent set of tools.) The following definition of expect is paraphrased from the manual page [5]:

expect *patlist1 action1 patlist2 action2 . . .*

> waits until one of the patterns matches the output of the current process, a specified time period has passed, or an end-of-file is found. If the final *action* is null, it may be omitted.
>
> Each *patlist* consists of a single pattern or list of patterns. If a pattern is matched, the corresponding action is executed. The result of the action is returned from expect. The exact string matched (or read but unmatched, if a timeout occurred) is stored in the variable expect_match. If *patlist* is eof or timeout, the corresponding action is executed upon end-of-file or timeout, respectively. The default timeout period is 10 seconds but may, for example, be set to 30 by the command `set timeout 30`.
>
> The following fragment is from a script that involves a login. abort is a procedure defined elsewhere in the script, while the other actions use Tcl primitives similar to their C namesakes.

```
expect "*welcome*"  break \
       "*busy*"     {print busy; continue} \
       "*failed*"   abort \
       timeout      abort
```

> Patterns are the usual C-shell-style regular expressions. Patterns must match the entire output of the current process since the previous expect or interact (hence the reason most are surrounded by the * wildcard). However, more than 2000 bytes of output can force earlier bytes to be "forgotten". This may be changed by setting the variable match_max.

expect actually demonstrates the best and worst of **expect**. In particular, its flexibility comes at the price of an occasionally confusing syntax. The pattern-lists can contain multiple patterns except for keyword patterns (e.g., eof, timeout) which must appear by themselves. This provides a guaranteed way of distinguishing them. However, breaking up the lists requires a second scan, which can interpret `\r` and `\n` as whitespace if not correctly quoted. This is exacerbated by Tcl providing two forms of string quoting: braces and double quotes. (If unambiguous, Tcl does not require strings to be quoted at all.) There is a separate section in the **expect** manual page to explain this complexity. Fortunately, a healthy set of examples seems to have held back complaints. Nonetheless, this aspect will be probably be revisited in a future release. For readability in this paper, scripts are presented as if double quotes sufficed.

Characters can be individually quoted with a backslash. Backslashes are also used to continue statements, which otherwise are terminated at the end of a line. This is inherent to Tcl. Tcl also continues scanning when there is an open brace or double-quote. In addition, semicolons can be used to separate multiple statements on a single line. This sounds confusing, but is typical of interpreters (e.g., **/bin/sh**). Nonetheless, it is one of the less elegant aspects of Tcl.

3. callback

It is surprising how little scripting is necessary to produce something useful. Below is a script that dials a phone. It is used to reverse the charges so that long-distance phone calls are charged to the computer. It is invoked as `expect callback.exp 12016442332` where the script is named **callback.exp** and +1 (201) 644-2332 is the phone number to be dialed.

```
# first give the user some time to logout
exec sleep 4
spawn tip modem
expect "*connected*"
send "ATD[index $argv 1]\r"
# modem takes a while to connect
set timeout 60
expect "*CONNECT*"
```

The first line is a comment. The second illustrates how a UNIX command with no interaction can be called. `sleep 4` will cause the program to block for four seconds, giving the user a chance to logout since the modem will presumably call back to the same phone number that the user is already using.

The next line starts **tip** using spawn so that **tip**'s output can be read by expect and its input written by send. Once **tip** says it is connected, the modem is told to dial the number. (The modem is assumed to be Hayes compatible, but it is easy to expand the script to handle others.) No matter what happens, **expect** terminates. If the call fails, it is possible for **expect** to retry, but that is not the point here. If the call succeeds, **getty** will detect DTR on the line after **expect** exits, and prompt the user with `login:`. (Actual scripts usually do more error checking.)

This script illustrates the use of command-line parameters, made available to the script as a list named **argv** (in the same style as the C language). In this case, element 1 is the phone number. The brackets cause the enclosed text to be evaluated as a command, and the result is substituted for the original text. This is similar to the way backquotes work in **csh**.

This script replaced a 60K program (written in C) that did the same thing.

4. passwd & conformance testing

Earlier, **passwd** was mentioned as a program that cannot be run without user interaction. **passwd** ignores I/O redirection and cannot be embedded in a pipeline so that input comes from another program or file. It insists on performing all I/O directly with a real user. **passwd** was designed this way for security reasons, but the result is that there is no way to test **passwd** non-interactively. It is ironic that a program so critical to system security has no way of being reliably tested.

passwd takes a username as an argument, and interactively prompts for a password. The following **expect** script takes a username and password as arguments, and can be run non-interactively:

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

The first line starts the **passwd** program, with the username passed as an argument. The next line saves the password in a variable for convenience. Like the shell, variables do not have to be declared in advance.

In the third line, 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 expect just waits until the pattern is found before continuing.

After receiving the prompt, the next line sends a password to the current process. The `\r` indicates a carriage-return. (All the "usual" C conventions are supported.) There are two expect-send 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** assumes it is interacting with a human that does not type consistently.

Lastly, the line `expect eof` searches for the end-of-file in the output of **passwd** and demonstrates the use of *keyword patterns*. Another such pattern is timeout, used to denote the failure of any pattern to match in a given amount of time. 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.

This script is sufficient to show the basic interaction of the **passwd** command. A more complete script would verify other behaviors. For example, the following script checks several other aspects of the **passwd** program. Complete prompts are checked. Correct handling of garbage input is checked. Process death, unusually slow response, or any other unexpected behavior is also trapped.

```
spawn passwd [index $argv 1]
expect eof                      {exit 1}     \
       timeout                  {exit 2}     \
       "*No such user.*"        {exit 3}     \
       "*New password:"
send "[index $argv 2]\r"
expect eof                      {exit 4}     \
       timeout                  {exit 2}     \
       "*Password too long*"    {exit 5}     \
       "*Password too short*"   {exit 5}     \
```

```
        "*Retype new password:"
send "[index $argv 3]\r"
expect timeout                    {exit 2}     \
        "*Mismatch*"              {exit 6}     \
        "*Password unchanged*}"   {exit 7}     \
        "\r\n"
expect timeout                    {exit 2}     \
        "*"                       {exit 6}     \
        eof
```

This script exits with a numeric indication of what happened. 0 indicates **passwd** ran normally, 1 that it died unexpectedly, 2 that it locked up, and so on. Numbers are used for simplicity – **expect** could just as easily pass back strings, including any messages from the spawned program itself. Indeed, it is typical to save the entire interaction to a file, deleting it only if the command under test behaves as expected. Otherwise the log is available for further examination.

This **passwd** testing script is designed to be driven by another script. This second script reads a file of arguments and expected results. For each set, it calls the first script and then compares the results to the expected results. (Since this task is non-interactive, a plain old shell can be used to interpret this second script.) For example, a data file for **passwd** could look like this:

```
passwd.exp     3     bogus   -              -
passwd.exp     0     fred    abledabl       abledabl
passwd.exp     5     fred    abcdefghijklm  -
passwd.exp     5     fred    abc            -
passwd.exp     6     fred    foobar         bar
passwd.exp     4     fred    ^C             -
```

The first field names the regression script to be run. The second field is the exit value that should match the result of the script. The third field is the username. The fourth and fifth fields are the passwords to be entered when prompted. The hyphen is just a placeholder for values that will never be read. In the first test, `bogus` is a username that is invalid, to which **passwd** will respond `No such user`. **expect** will exit the script with a value of 3, which also appears as the second element in the first line of the regression suite data file. In the last test, a control-C is actually sent to the program to verify that it aborts gracefully.

In this way, **expect** can be used for testing and debugging interactive software, such as required by IEEE POSIX 1003.2 (Shells and Tools) conformance testing. This is described in more detail by Libes [6].

5. rogue & pseudo-terminals

UNIX users are familiar with processes connected to other processes by pipes (e.g. a shell pipeline). **expect** uses ptys (pseudo-terminals) to connect spawned processes. Ptys provide terminal semantics so that programs think they are performing I/O with a real terminal.

As an example, the BSD adventure game **rogue** runs in raw mode, and assumes a character-addressable terminal exists at the other end of the connection. **expect** can actually be programmed to play **rogue** using the human interface that comes with it.

rogue is an adventure game which presents you with a player that has various physical attributes such as a strength rating. Most of the time, the strength is 16, but every so often – maybe one out of 20 games – you get an unusually good strength of 18. A lot of **rogue** players know this, but no one in their right mind restarts the game 20 times to find those really good configurations. The following script does it for you.

```
for {} {1} {} {
        spawn rogue
        expect "*Str: 18*" break \
                "*Str: 16*"
        close
        wait
}
interact
```

The first line is a for loop, with the same control arguments as in C. **rogue** is started, and then the strength checked to see if it is 18 or 16. If it is 16, the dialogue is terminated via close and wait (which respectively closes the connection to the pty and waits for the process to exit). **rogue** reads an end-of-file and goes away, after which the loop is restarted, creating a new game of **rogue** to test.

When a strength of 18 is found, control breaks out of the loop and drops down to the last line of the script. interact passes control to the user so that they can play this particular game.

Imagine running this script. What you will actually see is 20 or 30 initial configurations fly across your screen in less than a second, finally stopping with a great game for you to play. The only way to **play** rogue better is under the debugger!

It is important to realize that **rogue** is a graphics program which uses Curses. **expect** programmers must understand that Curses does not necessarily create screens in an intuitive manner. Fortunately, it is not a problem in this example. A future enhancement to **expect** may include a built-in terminal emulator in order to support the understanding of character graphics regions.

6. *ftp*

The first script actually written with **expect** did not print out `hello world`. Instead, it did something much more useful. It ran **ftp** without user interaction. **ftp** is a program which performs file transfer over TCP/IP networks such as the Internet. The ubiquitous implementation requires the user to provide input for all but the most simple uses.

The script below retrieves a file from a host using anonymous **ftp**. The hostname is the first argument to the script. The filename is the second argument.

```
spawn ftp [index $argv 1]
expect "*Name*"
```

```
send "anonymous\r"
expect "*Password:*"
send [exec whoami]
expect "*ok*ftp>*"
send "get [index $argv 2]\r"
expect "*ftp>*"
```

Dedicated programs have been written to perform *background ftp*. While they use the same underlying mechanism as **expect**, their programmability leaves much to be desired. Since **expect** provides a high-level language, you can customize it to your needs. For example, you can add:

- persistence – if the connection or transfer fails, you can retry every minute, hour, or even aperiodic intervals that depend on other factors such as user load.
- notification – you can be notified upon transmission via **mail**, **write** or any other mechanism of your choice. You can even be notified of failure.
- initialization – each user can have their own initialization file (e.g., **.ftprc**) in a high-level language for further customization, much like **csh** uses **.cshrc**.

expect could do many more sophisticated things. For example, it could use McGill University's Archie system. Archie is an anonymous **telnet** service that provides access to a database describing the contents of the entire Internet's anonymous **ftp** repositories. Using this, a script could ask Archie where a file is, and then download it to your system. This requires only a few more lines at the beginning of the **ftp** script above.

No known background-**ftp** programs provide even one of the features mentioned above, no less all of them. In **expect**, the implementation is trivial. Persistence requires a loop in the **expect** script. Notification is an `exec` of **mail** or **write**. An initialization file can be read with one command (`source .ftprc` does just the right thing) and can use any **expect** command.

Although these features can be added by hooks into existing programs, there is still no guarantee that everyone's needs will have been met. The only way to have such confidence is to provide a general-purpose language. A good solution would be to integrate Tcl, itself, directly into **ftp** and other applications. Indeed, that was the original intent of Tcl's design. Until this is done, **expect** provides much of the benefit of Tcl to many applications without any rewriting at all.

7. *fsck*

fsck is yet another example of a program with an inadequate user interface. **fsck** provides almost no way of answering questions in advance. About all you can say is "answer everything yes" or "answer everything no".

The following fragment shows how a script can automatically answer some questions "yes", and the rest "no". The script begins by spawning **fsck**, and then answering "yes" to two types of questions, and "no" to everything else.

```
for {} {1} {} {
      expect \
            eof                        break \
```

```
                "*UNREF FILE*CLEAR?"      {send "y\r"} \
                "*BAD INODE*FIX?"         {send "y\r"} \
                "*? "                     {send "n\r"}
}
```

In the next version, the two questions are answered differently. Also, if the script sees something it doesn't understand, it executes the interact command which passes control back to the user. The user keystrokes go directly to **fsck**. When done, the user can exit or return control to the script, here triggered by pressing the plus key. If control is returned to the script, it continues automated processing where it left off.

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

Without **expect**, **fsck** can be run non-interactively only with very reduced functionality. It is barely programmable and yet it is the most critical of system administration tools. Many other tools have similarly deficient user interfaces. In fact, the large number of these is precisely what inspired the original development of **expect**.

8. Controlling multiple processes: job control

expect's concept of job control finesses some of the usual implementation difficulties. Two issues are involved: The first is how **expect** handles *classic* job control, such as occurs when you press ^Z at the terminal. The second is how **expect** handles multiple processes.

The answer to the first issue is: Ignore it. **expect** doesn't understand anything about classic job control. For example, if you spawn a program and then send it a ^Z, it will stop (courtesy of the pty driver) and **expect** will wait forever.

In practice, however, this is not a problem. There is no reason for an **expect** script to ever send a ^Z to a process. It doesn't have to *stop* a process, per se. **expect** simply ignores a process, and turns its attention elsewhere. This is **expect**'s idea of job control and it works quite well.

The user view of this is as follows: When a process is started by spawn, the variable spawn_id is set to a descriptor referring to that process. The process described by spawn_id is considered *the current process*. (This descriptor is exactly the pty file descriptor, although the user treats it as an opaque object.) expect and send interact only with the current process. Thus, to switch jobs all that is necessary is to assign the descriptor of another process to the variable spawn_id.

Here is an example showing how job control can be used to have two **chess** processes interact. After spawning them, one process is told to move first. In a loop, a move is sent from one process to the other, and vice versa. The **read_move** and **send_move** procedures are left as an exercise for the reader. (They are actually very easy to write, but too long to include here.)

```
spawn chess                 ;# start player one
set id1 $spawn_id
expect "Chess\r\n"
send "first\r"              ;# force it to go first
read_move

spawn chess                 ;# start player two
set id2 $spawn_id
expect "Chess\r\n"

for {} {1} {} {
        send_move
        read_move
        set spawn_id $id1

        send_move
        read_move
        set spawn_id $id2
}
```

Some applications are not like a chess game where players alternate moves in lock step. The following script implements a *spoofer*. It will control a terminal so that a user will be able to log in and work normally. However, whenever the system prompts for either `password` or `login`, **expect** begins recording keystrokes until the user presses return. This effectively collects just the logins and passwords of a user without the usual spoofer problem of seeing `Incorrect password - try again`. Plus, if the user connects to another host, those additional logins will be recorded also![2]

```
spawn tip /dev/tty17             ;# open connection to
set tty $spawn_id                ;# tty to be spoofed

spawn login                      ;# open connection to
set login $spawn_id              ;# login process

log_user 0

for {} {1} {} {
        set ready [select $tty $login]
        case $login in $ready {
               set spawn_id $login
               expect  {"*password*" "*login*"} {
                      send_user $expect_match
                      set log 1
               } "*"        ;# ignore everything else
               set spawn_id $tty; send $expect_match
        }
```

2. The usual defense against a spoofer is to disallow write access so that the spoofer cannot open public terminals to begin with.

```
        case $tty in $ready {
              set spawn_id $tty
              expect "*\r*" {
                          if $log {
                                send_user $expect_match
                                set log 0
                          }
              } "*"
              if $log {
                    send_user $expect_match
              }
              set spawn_id $login; send $expect_match
        }
  }
```

The script works as follows. First connections are made to a **login** process and terminal. By default, an entire session is logged to the standard output (via send_user). Since this is not of interest, it is disabled by the command `log_user 0`. (A variety of commands are available to control exactly what is seen or logged.)

In a loop, select[3] waits for activity from either the terminal or the process and returns a list of spawn_ids with pending input. case executes an action if a value is found in a list. For example, if the string `login` appears in the output of the **login** process, the prompt is logged to the standard output and a flag is set so that the script will begin recording the user's keystrokes until a return is pressed. Whatever was received is echoed to the terminal. A corresponding action occurs in the terminal half of the script.

These examples have demonstrated **expect**'s form of job control. By interposing itself in a dialogue, **expect** can build arbitrarily complex I/O flow between processes. Multiple fan-out, multiplexed fan-in, and dynamically data-dependent process graphs are all possible.

In contrast, the shell makes it extraordinarily difficult just to read through a file one line at a time. The shell forces the user to press control characters (^Z, ^C) and keywords (fg, bg) to switch jobs. These cannot be used from shell scripts. Similarly, the shell running non-interactively does not deal with history and other features designed solely for interactive use. This presents a similar problem as with **passwd** earlier. Namely, that it is impossible to build shell scripts which regressively test certain shell behavior. The result is that these aspects of the shell will inevitably not be rigorously tested.

Using **expect**, it is possible to drive the shell using its interactive job control features. A spawned shell thinks it is running interactively, and will handle job control as usual. Not only does it solve the problem of testing shells and other programs that handle job control, but it also enables the shell to handle the job for **expect** when necessary. Processes to be manipulated with shell-style job control can be *backed* with a shell. This means that first a shell is spawned, and then a command is sent to the shell to start the process. If the process is suspended by, for example, sending

3. select calls poll() on USG systems and, in retrospect, should have been called something less biased and more meaningful.

a ^Z, the process stops and control returns to the shell. As far as **expect** is concerned, it is still dealing with the same process (the original shell).

Not only is **expect**'s approach flexible, it also avoids duplicating the job control software that is already in the shell. By using the shell, you get the job control of your choice since you can pick the shell to spawn. And should you need to (such as when testing), you really can drive a shell so that it thinks it is running interactively. This is also useful for programs that change the way they buffer output after detecting whether they are running interactively or not.

To further pin things down, during interact, **expect** puts the controlling terminal (the one **expect** was invoked from, not the pty) into raw mode so that all characters pass to the spawned process verbatim. When **expect** is not executing interact, the terminal is in cooked mode, at which time shell job control can be used on **expect** itself.

9. Using expect interactively

Earlier were shown scripts that are used interactively with interact. interact essentially gives a user free access to the dialogue, but sometimes finer control is desired. This can be achieved using expect which can read from the standard input just as easily as it reads from a process. A predefined spawn_id maps to the standard input and the standard output. Alternatively, the commands expect_user and send_user perform I/O with the standard input and the standard output without changing spawn_id.

The following script reads a line from the standard input for a given amount of time. This script (named **timed_read**) can be called from, for example, a **csh** script as `set answer = 'timed_read 30'`.

```
#!/usr/local/bin/expect -f
set timeout [index $argv 1]
expect_user "*\n"
send_user $expect_match
```

The third line accepts any newline-terminated line from the user. The last line returns it to the standard output. If nothing is typed before the timeout, nothing is returned.

The first line allows systems that support the #! magic to invoke the script directly (without saying `expect` before the script name) if its execute permission is set. Of course a script can always be invoked explicitly, as "`expect script`". Options preceded by a –c flag are executed as commands before any in the script. For example, an **expect** script can be traced without reediting by invoking it as `expect -c "trace ..." script` (where the ellipsis represents a tracing option).

Multiple commands may be strung together on a single script line or within braces, separated by semi-colons. Naturally, this extends to the –c argument. For example, the following command runs program **foo** for 20 seconds.

```
expect -c "set timeout 20; spawn foo; expect"
```

Once the timeout is set and the program is spawned, expect waits for either an end-of-file or the 20 seconds to pass. If the end-of-file is seen, the program has (almost certainly) exited, and expect returns. If the timeout has passed, expect returns. In either case **expect** exits, implicitly killing the current process.

It is educational to try and solve these last two examples without using **expect**. In both cases, the usual approach is to fork a second process that sleeps and then signals the original shell. If the process or read finishes first, the shell kills the sleeper. Passing pids and preventing the background process start message is a stumbling block for all but the most expert shell programmers. Providing a general approach to starting multiple processes this way complicates the shell script immensely. Invariably, the programmer writes a special-purpose C program.

expect_user, send_user, and send_error (for writing to the standard error) are frequently used in longer **expect** scripts which translate a complex interaction from a process to a simple one for the user. In [7], Libes describes how **adb** could be securely wrapped with a script, preventing a system administrator from needing to master the intricacies of **adb**, while at the same time dramatically lessening the likelihood of a system crash due to an errant keystroke.

A simpler example is automating **ftp** to retrieve files from a personal account. In this case, a password must be supplied. Storing the cleartext password in a file should be avoided even if the file permissions are heavily restricted. Supplying passwords as arguments to a script is also a security risk due to the ability of **ps** to retrieve them. A solution is to call expect_user at the beginning of the script for each password that the script must supply later. The password will be available to the script (and only to the script), even if it has to retry **ftp** every hour.

This technique is useful even if the information is to be entered immediately. For example, you can write a script which changes your password on every machine on which you have an account, whether or not the machines share a common password database (or even run UNIX). By hand, you might have to **telnet** to each machine and then enter the new password. With **expect**, you enter the password once and let the script do the rest of the work.

expect_user and interact can also be mixed in a single script. Imagine debugging a program that only fails after many iterations of a loop. An **expect** script could drive the debugger, setting breakpoints, running the program for the appropriate number of loops, and then returning control to the keyboard. It could also alternate between looping and testing for a condition, before returning control.

10. Programming expect interactively

expect may be programmed interactively. For example, if **expect** is run with no arguments, it prompts for commands. This is similar to what one normally does when interactively using a shell. This interactive mode may also be entered by pressing a user-defined string during interact.

Once prompted by the interpreter, you can type **expect** commands which are executed immediately. You can call defined procedures, perform job control, or even recursively invoke interact. For example, suppose you are running a script to automate **fsck**. You answer some of the questions yourself, and then decide that the rest should all be answered “yes”. You can escape from interact to the **expect** interpreter and invoke a procedure to answer the remaining questions without further interaction from you. This can be made as complex as you like.

The arguments to interact are actually string-action pairs. (The default action is to invoke the interpreter interactively.) This generalized mechanism can support all the usual styles of escapes. such as **tip**'s ~-prefixed commands or **csh**'s single-character job control keys. Actions may be any **expect** command. As an example, the following line maps the strings `~y`, `~a`, and the ^C and ^Z characters.

```
interact \
      ~y {yes} \
      ~a {send "[exec date]"; send_user "hello world"} \
      \Cc {exit} \
      \Cz {exec kill -STOP 0}
```

When `~y` is typed, a procedure called yes is invoked. This could further automate the **fsck** interaction just described, so that the user does not have to explicitly start the interpreter and type `yes`. `~a` invokes a more complex action. When typed, `hello world` is seen at the terminal and the current date is sent to the process as if the user had typed it. The other pairs exit or suspend an **expect** session while interacting with a spawned process. (With no map, the characters would be passed uninterpreted to the current process.) Appropriate maps can simulate **csh**-style job control or much fancier actions. For instance, ^Z could pass control to the interactive **expect** interpreter – analogous to what ^Z does in the shell – or it could change jobs to a spawned shell and resume the interaction.

An unrealistic but amusing application of character mapping is the following script which runs a shell with a Dvorak keyboard. For brevity, only lowercase letters are mapped.

```
proc dvorak {} {
      interact ~q {return continue} ~d {} \
            q {send '} w {send ,} e {send .} \
            r {send p} t {send y} y {send f} \
            u {send g} i {send c} o {send r} \
            p {send l} s {send o} d {send e} \
            f {send u} g {send i} h {send d} \
            j {send h} k {send t} l {send n} \
            x {send q} c {send j} v {send k} \
            b {send x} n {send b} , {send w} \
            . {send v} / {send z} ' {send -} \
            \; {send s} z {send \;} \
}

log_user 0
scan [exec printenv SHELL] "%s" shell
spawn $shell
log_user 1
send_user "~d for dvorak, ~q for qwerty (default)\n"
send_user "Enter ~ sequences using qwerty keys\n"
interact ~d dvorak ~q {}
```

This script has two interacts. The user switches between them by typing `~d` (for Dvorak) and `~q` (for qwerty). The Dvorak translation occurs in the procedure dvorak defined with proc. Within dvorak, an interact gives each character an action that corresponds to sending its Dvorak counterpart instead. Nothing has to be sent to the user, since the character will be echoed (if necessary) by the current process.

The `return continue` action for `~q` causes the Dvorak interact to return the value continue to its caller. interact's caller happens to be an earlier interact (at the bottom of the script) which evaluates the continue and literally continues. This isn't anything magical. They are just Tcl commands that are appropriately handled.

The script chooses the desired shell by examining the SHELL environment variable. Since **printenv** appends a newline to the end of its output, this has to be stripped off and is done here by scan – an equivalent to scanf in the C programming language.

This script is excessive and is not at all what this feature of interact was intended for. Nevertheless it works and demonstrates a number of interesting aspects.

11. Non-interactive programs are controlled differently

Some interactive programs have non-interactive alternatives. However, it is often the case that these alternatives are controlled in a way quite unlike the original interactive program. Thus, you need to learn two ways of doing things: interactively and non-interactively.

For example, suppose you want to locate a printer server. This is described by the `rm` value in the **printcap** file. Interactively, you might use an editor, or even, **more**, to search the file for the correct printer and then begin scanning for the `rm` field. To automate this, you must switch to a completely different program, such as **awk**.

Alternatively, you could just translate the interaction you were doing by hand into send/expect sequences. The following fragment does exactly this. It was used by a larger script that manipulated printers by running **lpc**, the interactive interface to the BSD line printer system.

```
spawn ed /etc/printcap
expect {*\n}     ;# discard character count
send "/$printer/\r"
for {} {1} {} {
        expect "*\r\n*:rm=*\n*" {
                # found rm, now get value
                set i [string first :rm= $expect_match]
                scan [range $expect_match [expr $i+4] end c] \
                        "%\[^:\]" server
                break
        } "*\r\n*\\\r\n" {
                # look at next line of entry
                send "\r"
        } "*\r\n*\n" {
                # no more lines in entry - give up
```

```
                    break
            }
    }
```

This script uses **ed** although any editor could be used. First **ed** is directed to search for the printer. Once the printer is found, returns are sent to get the successive lines until the value is either located or no more lines remain.

Using a specialized tool such as **awk** might seem like a better alternative, except if you aren't familiar with **awk**'s style of processing. While the same claim could be made about **expect**, this script illustrates the idea that (ignoring syntax differences) you can automate a procedure you know how to do interactively by simply translating it into send/expect sequences.

12. Is expect too fast?

The previous example demonstrated how **expect** can use an editor to read a file. **expect** has simpler ways of reading files. For instance the command `send [exec cat /etc/motd]` writes the contents of **/etc/motd** to the current process. Calling a UNIX program to read a file may not seem like a fast method but it is a lot faster than having a user type it in. In a window environment, cutting and pasting is an alternative, but this takes a large amount of time also. Realistically, blazing speed is hardly needed in a program that simulates users.

The speed of **expect** operations is described by Libes [2]. One side-effect not discussed is that **expect** can overrun input buffers designed for human typists. send supports a *slow* option (`send -s`) specifically to avoid this problem. It is controlled by parameters which describe the number of bytes to send atomically and a length of time to wait between each packet.

send also supports a simulation of actual human typing speed (`send -h`) according to a modified Weibull distribution [8], a common statistical tool to simulate interarrival times. The algorithm is driven by a random number generator and several user-chosen parameters. The parameters describe two average character interarrival times (default and word endings), minimum and maximum interarrival times, and a variability "shape". Errors are not simulated as this can be done by the user directly. Simplistic errors may be generated by embedding typing mistakes and corrections (if desired) in a send argument. A more sophisticated approach could use an expert system as a coroutine.

13. Security

The **passwd** script shown earlier was designed solely to be used for conformance testing. Many system administrators want such a script to embed in a comprehensive *adduser* script, which would set up everything that a generic new user needs including an account and password. Unfortunately, calling the **passwd** script from another script reopens the very problem that the **passwd** program was designed to solve. Passwords should not be used as arguments to programs because they can be seen by **ps** and similar programs.

The solution is to have the **expect** script generate the passwords directly. This closes the hole, while at the same time forcing the use of computer-generated passwords which are generally more difficult to guess than human-generated passwords.

This technique does not extend to programs such as **telnet**, **ftp**, **su**, etc., where a human really does need to provide the password. The solution is to have the **expect** script prompt for the password interactively via expect_user. In contrast to the program (or shell script) prompting when the password is needed, **expect** can prompt at the beginning of a script for all the passwords that will be needed. Even if the same password is used in several programs, the user need only enter it once since the script will remember it until it is needed.

Often, it is convenient to run such scripts in the background. Starting processes asynchronously from the shell, however, prevents them from reading keyboard input. Thus **expect** scripts must be started synchronously. The fork and disconnect commands are used later to move expect into the background.

For example, the following script reads the password, disconnects itself from the terminal, sleeps for one hour, and then goes on to execute commands that require a password.

```
system stty -echo              ;# disable echoing
send_user "Password: "
expect_user "*\n"
send_user "\n"                 ;# echo newline
system stty echo
scan $expect_match "%s" pass   ;# strip off terminating \n
if [fork]!=0 exit
disconnect
exec sleep 3600
spawn su
expect "*Password:*"
send "$pass\r"
# more commands follow
```

This script begins by disabling echo so that the password can be typed unseen. Unlike exec which manipulates its standard I/O so that it is accessible to **expect**, the system command does no manipulation, thereby allowing **stty** to affect the terminal.

fork literally causes **expect** to fork. Like the UNIX system call of the same name, it returns the child process ID to the parent. Since the parent has nothing else to do, it immediately exits. The shell will detect this as normal program termination. Meanwhile, disconnect breaks the association between the child process and the terminal so that the rest of the script can continue immune to the user logging out.

This paradigm provides a secure way of starting long-running background processes which require passwords. This works well with security schemes such as MIT's Kerberos system. In order to run a process authenticated by Kerberos, all that is necessary is to spawn **kinit** to get a ticket, and similarly **kdestroy** when the ticket is no longer needed.

Before **expect**, there was no way to achieve such results. The choice was either inflexibility or insecurity. **expect** has made this choice unnecessary, and given us the best of both worlds.

14. Conclusions

expect provides a means of automating interactive programs. There are a great many such programs in the UNIX domain that lack non-interactive alternatives. **expect** leverages off of these programs with only a small amount of programming effort.

expect solves a variety of problems with programs that 1) don't run non-interactively (**rlogin**, **telnet**); 2) "know" they're running interactively and change their behavior (**csh**, **rn**); 3) bypass stdio and open **/dev/tty** (**crypt**, **passwd**); 4) don't provide their full functionality non-interactively (**fsck**, **ftp**); or 5) don't provide the friendliest user interface (**adb**, **rogue**). All of the "new" non-interactive versions that result can now be usefully called from shell scripts because they can return meaningful error codes and no longer require user interaction.

expect provides help even when you want to run programs interactively. If they lack a programmable interface, you can partially automate the interaction and then share control. Of course, the ideal solution is to rewrite the application with a programmable front-end. For new applications, there is no excuse not to use Tcl. It is small, efficient, easy to use, and probably suffices for 90% of all tools. Building Tcl into an application will always be better than an after-the-fact solution like **expect**. But for tools which don't warrant the Tcl library, or are too old to be rewritten, **expect** is a fast solution.

expect is actually quite small. On a Sun 3, the current version is 64k. This includes the entire Tcl language. **expect** has few built-in functions. For example, **expect** doesn't have a communications protocol, nor does it know about sophisticated file access methods. It doesn't need to. It can invoke another program to do the work. At the same time, this gives you the flexibility of using any software you already have. Do you need to communicate with a serial line? Use **tip**, **cu**, or **kermit**. With a TCP socket? Use **telnet**. You make the choice.

This building block philosophy is very much in keeping with the UNIX tradition of hooking small programs together to build larger ones. In this respect, **expect** functions as a new kind of glue, much like the shell itself. Unfortunately, shell job control was designed only with interactive use in mind and cannot automatically control interactive processes. **expect**'s job control is generalized and has no such restriction. The two forms of job control do not interfere and can be used together.

While **expect** only runs on UNIX, it can be useful in managing non-UNIX sites as long as they are networked to a UNIX host. Via **telnet** or **tip**, a script can login and play its usual interactive games. My site has scripts that do exactly this on VMS and Symbolics Lisp machines. Our VMS wizards would rather avoid UNIX entirely, but they know a timesaver when they see it.

15. Acknowledgments

This work was supported by the National Institute of Standards and Technology (NIST) Automated Manufacturing Research Facility (AMRF). The AMRF is funded by both NIST and the Navy Manufacturing Technology Program.

Thanks to Scott Paisley who wrote the callback script. John Ousterhout is responsible for Tcl, without which **expect** would not have been written. John also critiqued **expect** as well as the first paper about it. I am indebted to him.

Several people made important observations or wrote early scripts while I was still developing the command semantics. Thanks to Rob Densock, Ken Manheimer, Eric Newton, Scott Paisley, Steve Ray, Sandy Ressler, Harry Bochner, Ira Fuchs, Craig Warren, Barry Warsaw, Keith Eberhardt, Jerry Friesen, and Dan Bernstein. Thanks to Mike Gourlay, Clem Cole, Andy Holyer, and Alan Crosswell for help in porting **expect** to various UNIX platforms. Thanks to Steve Simmons, Joe Gorman, and Corey Satten for fixing some of the bugs. Finally, thanks to K.C. Morris, Chuck Dinkel, Sue Mulroney, and the anonymous Computing Systems reviewers, who gave me extensive suggestions on improving this paper.

Certain trade names and company products are mentioned in order to adequately specify procedures and equipment used. In no case does such identification imply recommendation or endorsement by the National Institute of Standards and Technology, nor does it imply that the products are necessarily the best available for the purpose.

16. Availability

Since the design and implementation was paid for by the U.S. government, **expect** 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**ed anonymously as **pub/expect/expect.shar.Z** from **ftp.cme.nist.gov**. Request email delivery by mailing to `library@cme.nist.gov`. The contents of the message should be (no subject line) `send pub/expect/expect.shar.Z`.

As of August, 1991, over 2500 sites had retrieved **expect**.

17. References

[1] fsck, *UNIX Programmer's Manual*, Section 8, Sun Microsystems, Inc., Mountain View, CA, September, 1989.

[2] Don Libes, "expect: Curing Those Uncontrollable Fits of Interaction", *Proceedings of the Summer 1990 USENIX Conference*, Anaheim, California, June 11-15, 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] Don Libes, "expect User Manual", to be published as *NIST IR 744-91*, National Institute of Standards and Technology, Gaithersburg, MD.

[6] Don Libes, "Regression Testing and Conformance Testing Interactive Programs", *Proceedings of the Summer 1992 USENIX Conference*, San Antonio, Texas, June 8-12, 1992.

[7] Don Libes, “Using expect to Automate Systems Administration Tasks”, *Proceedings of the Fourth USENIX Large Installation Systems Administration (LISA) Conference*, Colorado Springs, Colorado, October 17-19, 1990.

[8] Norman Johnson, Samuel Kotz, “*Continuous Univariate Distributions*”, Vol. 1, Houghton Mifflin Co, New York, NY, 1970.