We spend a lot of our lives extracting information from a mass of data. In our professional lives this is often a case of scanning files or program output for an item of interest. For me, this frequently starts with grep, and ends when I've reduced that mass of data to just few interesting items, and perhaps correlated it with some other data.

The student computer lab where I'm teaching this semester is expected to be used only by students who are currently on-line. However, what with one glitch and another, students frequently end up leaving processes running after they've logged out. Since these tasks chew up resources to no good use, we try to find and kill these tasks.

The algorithm for finding the unwelcome tasks is pretty simple: use `who` to find out who is currently logged into a machine, run `ps` to see what tasks are active, and look for tasks that aren't owned by root, lp, daemon, or the currently logged in students.

As you might guess, Tcl has a useful set of commands for reading files or program output, manipulating text strings and reporting results to automate this process.

I'll introduce some Tcl commands for I/O, and string manipulation, and then show how the application looks.

Tcl I/O commands follow the familiar convention of creating a handle to access the data stream. This handle (called a `channel` in Tcl) may be used to access a file, device, pipe to another application, or a socket. A channel to a file, device or pipe is created with the `open` command. For a socket channel, the `socket` command is used. I'll discuss the `socket` command in a future article.

Syntax: open *streamName ?access? ?permissions?*

streamName	By default, the name of a file to open. If the first character of the *streamName* is a pipe symbol "|", then the rest of the name is a program to run attached to a pipe.
?access?	The access method: "r" for read, "w" for write, "a" for append. Or a list of POSIX mnemonics including `RDONLY WRONLY RDWR APPEND CREAT EXCL NOCTTY NONBLOCK TRUNC` The default is "r" (`RDONLY`).
?permissions?	When a file is created, you can declare the permissions mask in numeric form. Tcl supports octal numbers, allowing you to set the modes to values like 0666.

Tcl will substitute a command within square brackets with the result of evaluating that command. Thus, we open a channel to a file with a command like:

```
set inFile [open /etc/passwd "r"]
```

or, to read input from another program:

```
set inFile [open "!who" "r"]
```

Tcl uses the commands `gets`, `read` and `puts` for I/O. The `gets` command is useful for line-by-line input, while `read` is useful for block reads. The `puts` command will write a single line to a channel.

Syntax: gets *channel ?variableName?*

`gets` reads a line of input from the given channel.
If no `variableName` is present, `read` returns the string of input data.
If a `variableName` is present, the input data is placed in the variable with that name, and the number of characters read is returned.

The Tcl `gets` command doesn't generate an error if you try to read past a file's End-Of-File, it just returns a length of -1. Thus, you can read lines from a channel with a loop like:

```
 while {[gets $infl line] >= 0} {
    # Do Stuff to $line
}
```

You can also check for EOF with the `eof` command, which returns true when all of the data from a channel has been read. If you use `eof` the read-loop would resemble:

```
 while {![eof $infl]} {
    set len [gets $infl line]
    # Do Stuff to $line
}
```

Now that data has been read, it's time to process it. The `string` command has several subcommands for manipulating strings, but the 'find orphan processes' task only uses a few of these.

Syntax:`string wordend` *string index*

Returns the index of the character just after the last character in the word that includes the position *index*.

Syntax:`string trim` *string ?trimChars?*

Trims off all leading and trailing instances of the characters defined by *trimChars*. If *trimChars* isn't defined, then `string trim` trims off whitespace.

Syntax:`string range` *string start end*

Returns the characters in *string* between the *start* and *end* index markers.

Syntax:`string first` *string1 string2*

Returns the index of the first occurrance of *string1* in *string2*.

With two of those commands, we can extract the first word from the `who` output (the usernames of the folks currently logged in) with a command like:

```
set name [string range $line 0 [string wordend $line 0]]
```

We can extract the UID field from the `ps` output with a line like:

```
set uid [string trim [string range $line 5 14]]
```

which will extract the characters between the 5'th and 14'th position in the string, and then strip off any spaces.

Finally we check that this UID is not owned by someone currently logged in with:

```
string first $uid $namelist
```

If the name in `$uid` is not in the string `$namelist`, `string first` will return -1. If `$uid` is in `$namelist` then `string first` will return a value >= 0.

That explains how to get the names, and find out what lines from `ps` aren't owned by someone currently logged in, but how do we get the strings to process?

Once the data has been read and searched, it's time to format and report the results.

The Tcl `format` is equivalent to `sprintf`. It accepts a `printf`-like format string and a set of arguments, and returns a formatted string.

We can extract the portions of the PS output that we are interested in, and make a new display with code like:

```
    set id [string trim [string range $line 5 14]]
    set pid [string trim [string range $line 14 20]]
    set cmd [string trim [string range $line 83 end]]
    puts [format "%12s - %5d - %s" $id $pid $cmd]
```

The `format` command returns a string which is sent to the standard output device with the `puts` command.

This is the '90s, so we should display our results in a GUI (whether it's appropriate or not).

The simplest way to report a string of results like this is to use the Tk text widget. The text widget is a powerful tool that supports multiple fonts, colors, scrolling, editing, and more. You can insert images and other windows into a text window, and can bind actions to events that happen on single characters or large sections of text.

Using the text widget to just display this output is a bit like using a shotgun on a mosquito, but one of the freebies we get with the text widget is the ability to scan up and down the lines of text with ^N and ^P as if we were editing in emacs. This saves me from having to discuss scrollbars in this column.

The text widget has a decent set of defaults, so we could create the widget with a simple command like `text .t`. By default, a text window is 80 characters wide, and 24 lines tall. To make life a little more interesting, let's set the size explicitly, and use a slightly larger than normal font.

```
set txt [text .t -font {courier 18 bold} -height 10 -width 90
```

Now, instead of using the `puts` call to display the results, we can use the text widget's `insert` subcommand.

Syntax: *`textWidget`* `insert` *`index text`*

Insert the *text* at location `index` in the text widget.

The code to run on each machine and find orphaned processes is shown below. In actual fact, while the guts of the code I use resembles this, I actually run an expect script that logs into each machine on the local network and looks for orphaned processes. It reports the data as simple text strings. However, expect is a topic for another column.

```
# Open a text window for display.


set outputWindow [text .t -height 10 -width 90 \
     -font {courier 18 bold} -disable ]


pack $outputWindow


# Initialize the namelist with the names of users
#  that we know will be online (root, daemon, lp),
#  and add "UID" to cleverly remove the header
#  from consideration.


set namelist "root daemon lp UID"


# Run who and read the input from the who command.


set infl [open "|who" ]


# The gets call will return -1 when it hits an EOF.
#  Read the lines, extract the user name,
#  and if the username isn't already in our list, add it.


 while {[gets $infl line] >= 0} {
    set name [string range $line 0 [string wordend $line 0]]
    if {[string first $name $namelist] < 0} {
        set namelist "$name $namelist"
    }
}


# close the file, we're done with it.
catch {close $infl}


# Invoke ps, and read the input.


set infl [open "|ps -elf"]


# This time, we'll use the eof command to check
#  for end of file.


while {![eof $infl]} {
    set len [gets $infl line]
```

```
    # extract the user id from the line of data.

    set id [string trim [string range $line 5 14]]

    # If the id is not in our namelist, we have a hit.
    # Get the pertinent data and update the window.
    if {[string first $id $namelist] < 0} {
        set pid [string trim [string range $line 14 20]]
        set cmd [string trim [string range $line 83 end]]
        $outputWindow insert end  \
            [format "%12s - %5d - %s\n" $id $pid $cmd]
    }
}
```