screen: Windows that Follow You

By Jerry Peek

How can you start a job from one system and finish it on another? For example:

➤ You start an IRC (chat) session in a terminal window on one computer, but you want to continue it from another system — without having to `/leave` (and lose your place) and try to `/join` again later.

➤ Or, you're at home and you connect to your office system and start work. But you have to leave home for the office before you've finished, and you need to pick up where you left off once you're at the office. Or, perhaps your modem hangs up, the network connection goes down, or the firewall times out your connection and you need to restart with no pain.

The handy *screen* utility lets you resume a terminal session without missing a keystroke. (*screen* doesn't work with GUI applications.) Read on for some surprising ways to use *screen*. Its clever use of processes, *ttys*, the environment, and more, can also give users and programmers insight into how "the guts" of Linux and Unix work.

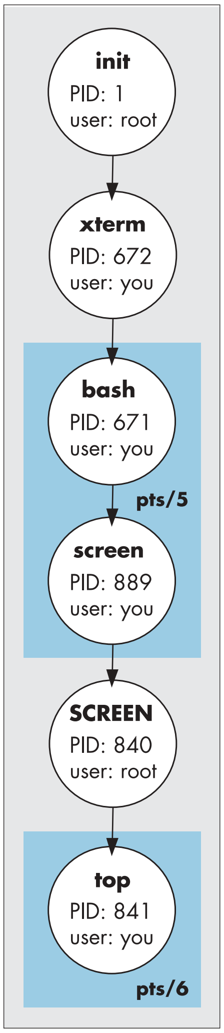

FIGURE ONE: Processes and *ptys* with one *screen* window open

screen Then and Now

In the mid '80s, when Oliver Laumann released *screen* version 1, most users had single terminals with no windows — and no Internet. These days, *screen* still works on dedicated terminals, but it's also very useful in terminal windows (under the X Window System) and over network connections.

What's made *screen* unique, then and now, is that it *maintains the state* of one or more *terminal window sessions* on a particular host. (These "terminal windows" are *character terminal* interfaces, like the old VT100 — not GUI windows.) Each session can have many terminal windows — or just one (and we'll see why you'd want this). You can *detach* from a session and then, later, *reattach* to that session, from the same system or from another one. This ability makes *screen* especially useful when it's used on a server with network connections.

screen, Inside and Out

How does *screen* work? Let's start with a simple example. (We'll use *screen* version 3.9.13, setuid *root*, but all of the examples shown should also work non-setuid and with other versions.) In a terminal — or a terminal window, like *xterm* — type the command name *screen* followed by another command you want to run under *screen*'s control.

For example, *top*, the system monitoring program, is a good choice because it keeps running and updating its display. (If you haven't used *top* before, start it now, watch it for a little while, then type `q` to quit.) Use the "delay" option — which is either `-d 60` or `-s60`, depending on your version — to make *top* refresh its display every sixty seconds. If your version of *top* uses option `-s` to set "secure" mode (instead of setting the delay time), use it too. Here's what you might type:

```
$ screen top -d 60 -s
```

FIGURE TWO: *ps* output while one *screen* window is open

```
$ ps -ew —forest -Oc
    ...
672  tty2  xterm
674  pts/5  \_ bash
839  pts/5      \_ screen top -d60 -s
840  ?              \_ SCREEN top -d60 -s
841  pts/6              \_ top -d60 -s
```

FIGURE THREE: *ps* output with two *screen* windows open

```
$ ps -ew —forest -Oc
    ...
672  tty2  xterm
674  pts/5  \_ bash
839  pts/5      \_ screen top -d60 -s
840  ?              \_ SCREEN top -d60 -s
841  pts/6              \_ top -d60 -s
922  pts/7              \_ bash
924  pts/7                  \_ vi afile
```

A WINDOW'S ENVIRONMENT

If you're at a shell prompt *in a window that's under* screen*'s control*, you can open a new window by typing *screen* and the command line you want to run in that new window. That's handy if you want to run a single command, like *top*, in a new window — it's quick to type and it doesn't start a new shell, so it's efficient. But how does it work? When you run a new instance of *screen*, how can it find out that it's running from a shell that's under *screen*'s control? The answer comes from understanding more of what's happening "inside" each window process.

When a Linux process (the *parent*) starts a new process (the *child*), the parent's environment is copied to the child. That includes environment variables. And *screen* sets several environment variables that are worth knowing about. If you type the Linux command *env* at a prompt in any window, you'll see the environment variables set there. Let's try it in the second window, where *vi* was running:

```
$ env
...
STY=840.pts-5.jpeek
TERM=screen
TERMCAP=SC|screen|VT 100...
WINDOW=1
```

The `TERM` and `TERMCAP` variables are from the Unix terminal capability system, which tells any program running in that terminal (such as *vi*) how to clear the screen, move the cursor, and so on. The `STY` variable points to the backend *SCREEN* process; notice that its value has the PID (`840`) and the number of the pty where it originally started (`pts-5`). So, a new *screen* process can check its environment to see if the backend *SCREEN* process is its ancestor.

Finally, `WINDOW` is the number that *screen* assigns to each of its windows: window 0 is the first window, 1 is the second, and so on. You can switch to any of the windows by typing `C-a` *`n`*, where *n* is the window number. (So, to see the window running *top*, type `C-a 0`.)

The command `C-a N` shows the current window number and title (in an *xterm* title bar, or in the message area at the top of a window). You can also use the `WINDOW` variable to put the window number in your shell prompt. If you use the *tcsh* or *csh* shell, put the code shown below into your *.tcshrc* or *.cshrc* file.

```
if ($?prompt) then
   # If shell is running under screen, put
   # window number before the % prompt:
   if ($?WINDOW) then
      set prompt = "screen${WINDOW}% "
   else
      set prompt = "% "
   endif
endif
```

Or, in a Bourne-type shell like *bash*, the following code will do the job in a single line of your *.bashrc*:

```
# If this shell is running under screen,
# put window number before the $ prompt:
PS1="${WINDOW:+screen$WINDOW}$ "
```

If the `WINDOW` environment variable is set to `0`, for instance, the prompt is set to `screen0%` in C-type shells and `screen0$` in Bourne-type shells. If `WINDOW` isn't set, then the shell isn't running under *screen*, so the prompt is simply `%` or `$`. (The Bourne shell code uses the shell's `${`*`var`*`:+`*`value`*`}` parameter-substitution operator.)

At this point, you've got the processes and *ptys* shown in *Figure One*.

When you're using a terminal window (like the xterm here — as opposed to an old-fashioned "real" terminal), *xterm* uses a pty to manage the display. *screen* also creates a pty for each "window" that it manages. Right now, you're using two ptys: pts/5, which is the xterm display, and pts/6, which is the top display.

(A *tty* is the original Unix terminal interface device. A *pty*, or "pseudo-tty", is a virtual terminal. A pty has a screen buffer that can be manipulated like a terminal, and a number like *pts/5* that identifies it. You can see any terminal's number by running the Linux command *tty* from a shell prompt. The pty is another important Linux feature that *screen* uses.)

You can see the processes and ptys and their associations with one another with the *ps* command. Its `—forest` option arranges processes in a tree, as *Figure Two* shows.

Notice that the backend *SCREEN* process doesn't have a tty. (This is also true of processes like daemons, *cron* and *at* jobs, and most other Linux processes that don't need a user interface.) This backend process is the parent of all the other "windows" (processes) that *screen* manages. If you detach your terminal (here, *pts/5*) and log out, then, once you reattach, *SCREEN* can use *pts/6* to refresh your new terminal with exactly what was there when you left. (We'll soon see how to detach and reattach.)

screen also watches the keystrokes you type. It passes every keystroke through to the child process running in the current window — until it sees the special *escape character*, which is CTRL-A (which we'll write, Emacs-style, as `C-a`, and which you make by holding down the CTRL key and pressing the A key). After `C-a`, *screen* looks for exactly one more character, which is a *screen* command.

For instance, `C-a c` creates a new window process. (If you're following along at a terminal, please do that now.) The current window (the *top* load-average monitor) isn't shown anymore, but *screen* continues

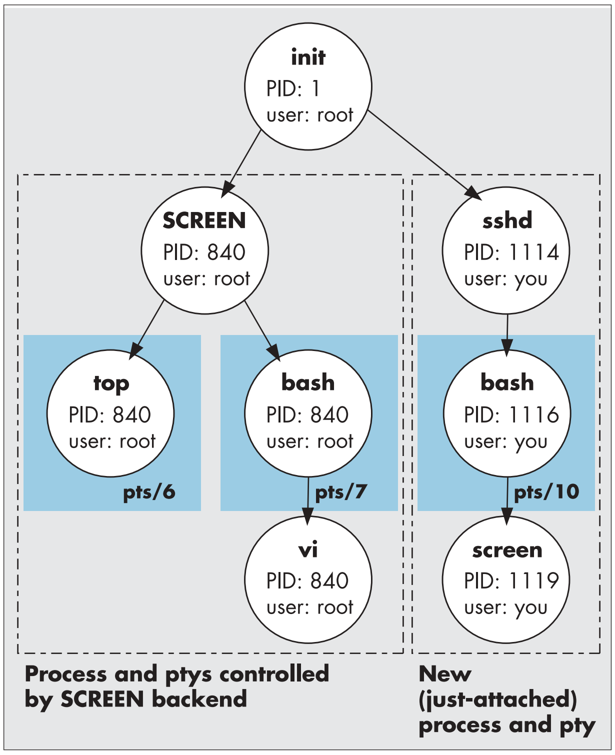

FIGURE FOUR: Processes and ptys after logging out, in, and reattaching

POWER TIP: Remembered Directories, continued

The February 2003 Power Tip (available online at http://www.linux-mag.com/2003-02/power_01.html) described `cd -` and directory stacks. Many shells (like *bash*) also let you use these directories *as arguments to other commands*. In these shells, the parameter `~-` (tilde dash) expands to the previous directory (before you typed `cd -`). The parameter `~1` is the top directory on the stack, `~2` is the second, and so on.

For example, here's how to move the file *foo* from the current directory into the directory */prj/baz* and to print the file */prj/bar/afile*:

```
$ dirs
/home/zoe /prj/bar /prj/baz
$ mv foo ~2
$ lpr ~1/afile
$ dirs
/home/zoe /prj/bar /prj/baz
```

If your shell (like the original *csh*) doesn't support these parameters, you can add aliases for *pushd* and *popd* that store the output of *dirs* into variables or into a shell array named *d*. The array setup is easier:

```
alias setdirs  'set d = (`dirs`)'
alias pushd    '""pushd \!*; setdirs'
alias popd     '""popd \!*; setdirs'
```

Then `$d[2]` gives the top of the stack, `$d[3]` the second, and so on.

to "remember" its state, even if it changes while it's not being displayed. In the new window, you should see a shell prompt. There you can start another program — for instance, a text editor like *vi*.

Running *ps* again shows the new processes. Now, both top-level window processes are children of *SCREEN*, and *vi* is a child of the *bash* shell. See *Figure Three*.

`C-a C-a` switches to the previously-displayed window — in this case, your *top* session. As before, *vi* will keep running; you can switch to it anytime to do more editing with `C-a C-a`.

If you want to be notified whenever the text changes in a "background" window, use `C-a M` to set *monitoring* of that window. Try that while you display the window running *top*, then change to the other window. Every minute, you should see `Activity in window 1` in the titlebar or message area of window 0. This is handy for monitoring a system log (with `tail -f`, say) while you're doing something else.

Detaching, Re-attaching

If you quit one of the processes under *screen*'s control (for instance, by typing `q` to *top* or typing `exit` at a shell prompt), that window will close. When its last window closes, *screen*, and its backend process, *SCREEN*, terminate.

If some *screen* windows are open, though, you can also leave *screen* temporarily. Here are two ways:

➤ You can suspend *screen* by typing `C-a z` or `C-a C-z` from any window. You can restart it (*before* you log out!) by typing the shell's *fg* command. This uses Linux job control.

➤ You can detach *screen* from your current terminal. The *SCREEN* backend process and the ptys it manages keep running. You can log out and log in again later — even days or weeks later (as long as the system isn't rebooted) — and reattach the *SCREEN* backend process to your *new* terminal. To detach *screen* from your terminal, type `C-a d`. You should see the message `[detached]`. Now you'll be back at a prompt from the shell where you first started (or attached) *screen*.

For example, if you're monitoring a long-running industrial process from a tty on your office workstation, and you're afraid that it won't finish before you have to go home, start that processes under *screen* control. When you leave, use `C-a d` to detach. Now you can turn off your display (but not your CPU!), go home, connect to your workstation by (for instance) *ssh*, and reattach. It's that easy!

To reattach, type the command `screen -r` at a shell prompt. Your window should appear just as it was when you left it — unless its contents have changed in the meantime, of course. You can see previous lines of the display by using *screen*'s scrollback history.

A single Linux system can have thousands of processes — including lots of *SCREEN* backend processes. So how can your *screen* find the right one to attach? Depending on your system's configuration, there'll be a subdirectory somewhere — *.screen* under your home directory, for instance — with a FIFO (named pipe) connected to the backend process. (Your version of *screen* may use a socket instead of a FIFO.) You can list running backends with the command `screen -list`.

Figure Four shows the complex setup after you've detached, logged out, logged in, and reattached. The backend *SCREEN* process and its windows are still running, with their same ptys and processes. When you type *screen* from your new shell prompt (which happens to be in *pts/10* this time, instead of *pts/5*), it finds the backend process and repaints your terminal from the active window's pty (here, that's *pts/6* or *pts/7*).

More Ways to Use *screen*

screen is useful when you're planning to detach and reattach. But it's also useful for a host of other things, such as protecting against network failures and capturing screen output.

➤ *screen* can keep a process alive if your network connection to it is broken. For instance, if you started Emacs on your server from under *screen*, and your network connection is broken, you can simply re-login to the server and reattach.

➤ *screen* can also copy and paste, paste copied text to a file, and write the scrollback history to a file — all without using a mouse! *screen* also lets you select text more precisely than a mouse typically does.

➤ If you're running a *screen* session on one system (at home, for example) and you leave without detaching, you may still be able to attach from another system with `screen -x`, which sets the multi-display mode.

There's much more to know about *screen*. Check its extensive manual page... and prepare to be impressed!

Thanks to Kimmo Suominen of Global Wire for some great *screen* tips. In July, we'll dig into email transport and processing tools.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers at jpeek@jpeek.com.