On UNIX® systems, each system and end-user task is contained within a process. The system creates new processes all the time and processes die when a task finishes or something unexpected happens. Here, learn how to control processes and use a number of commands to peer into your system.

At a recent street fair, I was mesmerized by the one-man band. Yes, I am easily amused, but I was impressed nonetheless. Combining harmonica, banjo, cymbals, and a kick drum -- at mouth, lap, knees, and foot, respectively -- the veritable solo symphony gave a rousing performance of the Led Zeppelin classic "Stairway to Heaven" and a moving interpretation of Beethoven's Fifth Symphony. By comparison, I'm lucky if I can pat my head and rub my tummy in tandem. (Or is it pat my tummy and rub my head?)

Lucky for you, the UNIX® operating system is much more like the one-man band than your clumsy columnist. UNIX is exceptional at juggling many tasks at once, all the while orchestrating access to the system's finite resources (memory, devices, and CPUs). In lay terms, UNIX can readily walk and chew gum at the same time.

This month, let's probe a little deeper than usual to examine how UNIX manages to do so many things simultaneously. While spelunking, let's also glimpse the internals of your shell to see how job-control commands, such as Control-C (terminate) and Control-Z (suspend), are implemented. Headlamps on! To the bat cave!

**A real multitasker**

On UNIX (and most modern operating systems, including Microsoft® Windows®, Mac OS X, FreeBSD, and Linux®), each computing task is represented by a process. UNIX runs many tasks seemingly at the same time because each process receives a little slice of CPU time in a (conceptually) round-robin fashion.
A process is something of a container, bundling a running application, its environment variables, the state of the application's input and output, and the state of the process, including its priority and accumulated resource usage. Figure 1 pictures a process.

**Figure 1. A conceptual model of a UNIX process**

If it helps, you can think of a process as its own sovereign nation, with borders, resources, and gross domestic product.

Each process also has an **owner**. Tasks you initiate -- your shell and commands, say -- are typically owned by you. System services might be owned by special users or by the superuser, root. For example, to enhance security, the Apache HTTP Server is typically owned by a dedicated user named **www**, which provides access to the files the Web server needs, but no others.

Ownership of a process might change but is otherwise strictly exclusive. A process can have only one owner at any given time.

Finally (and simplifying for this introduction), each process has **privileges**. Typically, a process's privileges are commensurate with those of its owner. (For instance, if you can't access a particular file from your command-line shell, programs you launch from the shell inherit the same limitation.) An exception to this inheritance rule, where a process might acquire greater privileges than its owner, is an application with the special **setuid** or **setgid** bit enabled, as shown by `ls`.

The setuid bit can be set using `chmod u+s`. setuid permissions look like this:

```
$ ls -l /usr/bin/top
-rwsr-xr-x 1 root  wheel 83088 Mar 20 2005 top
```

The setgid bit can be set using `chmod g+s`:

```
$ ls -l /usr/bin/top
-r-xr-sr-x 1 root tty 19388 Mar 20 2005 /usr/bin/wall
```

A setuid process, such as launching `top`, runs with the privileges of the user who owns the file. Hence, when you run `top`, your privileges are promoted to those of **root**. Similarly, a setgid process runs with the privileges associated with the group owner of the file.
For instance, on Mac OS X, the `wall` utility -- short for "write all," because it writes a message to every physical or virtual terminal device -- is setgid `tty` (as shown above). When you log in and are assigned a terminal device to type in (the terminal becomes standard input for your shell), you're made the owner of the device, and `tty` becomes the group owner. Because `wall` runs with the privileges of group `tty`, it can open and write to every terminal.

**Taking inventory**

Like all other system resources, your UNIX system has a finite, albeit large pool of processes. (In practice, a system almost never runs out of processes.) Each new task -- say, launching `vi` or running `xclock` -- is immediately allocated a process from the pool. On UNIX systems, you can view one or more processes using the `ps` command.

For example, if you want to see all the processes you own, type `ps -w --user username`:

```
$ ps -w --user mstreicher
```

You can view the entire list of processes using `ps -a -w -x`. (The format and specific flags of the `ps` command vary from UNIX flavor to UNIX flavor. See the online documentation for your system to find specifics.) `-a` selects all processes running on a tty device; `-x` further selects all processes not associated with a tty, which typically includes all the perpetual system services, such as the Apache HTTP server, the `cron` job scheduler, and so on; and `-w` shows a wide format, useful for seeing the command line or full pathname of the application associated with each process.

`ps` has a legion of features, and some versions of `ps` even allow you to customize the output. For example, here is a useful custom process listing:

```
$ ps --user mstreicher -o pid,uname,command,state,stime,time

PID USER     COMMAND          S STIME     TIME
14138 mstreic  sshd: mstreicher S 09:57 00:00:00
14139 mstreic  -bash            S 09:57 00:00:00
14937 mstreic  ps --user mstrei R 10:23 00:00:00
```

- `-o` formats output according to the order of the named columns. `pid`, `uname`, and `command` are process ID, user name, and command, respectively. `state` reflects the process state, such as sleeping (s) or running (R). (More on process state in a moment.) `stime` shows when the command started, and `time` shows how much CPU time the process has consumed.

**Daddy, where do processes come from?**

On UNIX, some processes run from system boot to shutdown, but most processes come and go rapidly, as tasks start and complete. At times, a process can die a premature, even horrible death (say, due to a crash). Where do new processes come from?

Each new UNIX process is the `spawn` of an existing process. Further, each new process -- let's call it the "child" process -- is a clone of its "parent" process, at least for an instant, until the child continues execution independently. (If each new process is the offspring of an existing process, that begs the quandary, "Where does the first process come from?" See the sidebar below for the answer.)
The chicken and the egg

Some debates are perennial: To be or not to be? Coke or Pepsi? PC or Mac? Then, of course, there's the age-old quandary, "Which came first: the chicken or the egg?"

If each new UNIX process is spawned from an existing, running process, where does the first process come from? The answer: The UNIX kernel spawns the first process during the boot sequence.

The first process is called, appropriately enough, init, and the genealogy of all other system processes can be traced back to init. In fact, init's process number is 1. You can find the status of init by typing ps -l 1:

F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
4 S 0 1 0 0 68 0 - 373 select ? 0:02 init[2]

As you can see, the owner (UID) of init is 0 (root). Unlike every other process in the system, init doesn't have a parent process -- the Parent Process ID (PPID) is 0.

Figures 1-4 detail the spawning, uh, process:

1. In Figures 2 and 3, Process A is running a program represented by the blue box. It runs the instructions numbered 10, 11, 12, and so on. Process A has its own data, its own copy of the program, its set of open files, and its own collection of environment variables, which were initially captured when Process A sprang into existence.

**Figure 2. Process A running code**

2. In UNIX, the fork() system call (so named because it's a call, or request, for operating system assistance) is used to spawn a new process. When Program A executes fork() in Instruction 13, the system immediately creates an exact clone of Process A, named Process Z. Z has the same environment variables as A, the same memory contents, the same program state, and the same files open. The state of Processes A and Z immediately after Process A spawns process Z is shown in Figure 3.
3. At inception, Process Z begins execution at the same place where Process A left off. That is, after inception, Process Z begins execution at Instruction 14. Process A continues execution at the same instruction.

4. Typically, the programming logic at Instruction 14 tests whether the current process is the child or parent process -- that is, Instruction 14 in Process Z and Instruction 14 in Process A separately determine if its process is the progeny or progenitor. To differentiate, the `fork()` system call returns 0 in the progeny but returns the process ID of Process Z to the progenitor.

5. After the previous test, Process A and Process Z diverge, each taking a separate code path, as if both came to a fork in the road and each took a distinct branch. The process of spawning a new process is more often called forking, given the metaphor of two travelers reaching a fork in the road. Hence, the system call is named `fork()`.

After the fork, Process A might continue running the same application. However, Process Z might immediately choose to metamorphose to another application. The latter operation of changing what program is running with a process is called execution, but you can think of it as reincarnation: Although the process ID remains the same, the instructions within the process are replaced entirely with those of the new program. Figure 4 shows the state of Process Z some time later.

**Figure 3. Process A spawns a clone of itself**

**Figure 4. Process Z is now independent of its progenitor, Process A**
Forking around

You can experience forking right from the comfort of your private command line. To begin, open a new xterm. (You likely now realize that xterm is its own process and, within xterm, the shell is a separate process spawned by xterm). Next, type:

```
ps -o pid,ppid,uname,command,state,stime,time
```

You should see something like this:

<table>
<thead>
<tr>
<th>PID</th>
<th>PPID</th>
<th>USER</th>
<th>COMMAND</th>
<th>S</th>
<th>STIME</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>16351</td>
<td>16350</td>
<td>mstreic</td>
<td>-bash</td>
<td>S</td>
<td>11:23 00:00:00</td>
<td></td>
</tr>
<tr>
<td>16364</td>
<td>16351</td>
<td>mstreic</td>
<td>ps -o pid,ppid,u</td>
<td>R</td>
<td>11:24 00:00:00</td>
<td></td>
</tr>
</tbody>
</table>

According to the PPID fields in this list, the ps command is a child of the bash shell. (The hyphen in -bash indicates that the shell instance is a login shell.) To run ps, bash forks to create a new process; the new process reincarnates itself using execution, turning into a new instance of ps.

Here's another experiment to try. Type:

```
sleep 10 & sleep 10 & sleep 10 & ps -o pid,ppid,uname,command,state,stime,time
```

You should see something like this:

```
$ sleep 10 & sleep 10 & sleep 10 &
[1] 16843
[2] 16844
[3] 16845
$ jobs
[1]  Running  sleep 10 &
[2]  Running  sleep 10 &
[3]  Running  sleep 10 &
```

The command line spawns four new processes. Typing ampersand (&) after each sleep command runs each of those commands in the background, or in parallel with the shell. ps is another spawned process, but it's running in the foreground, preventing the shell from running another command until it terminates. Again, all four processes are the spawn of the shell, as shown by the values of PPID. The three sleep commands are marked s, because none of the process are consuming resources while they're sleeping.

For convenience, the shell keeps track of all background processes it spawns. Type jobs to see a list:

```
$ sleep 10 & sleep 10 & sleep 10 &
[1] 16843
[2] 16844
[3] 16845
$ jobs
[1]  Running  sleep 10 &
[2]  Running  sleep 10 &
[3]  Running  sleep 10 &
```
Here, the three jobs are labeled 1, 2, and 3 for convenience. The numbers 16843, 16844, and 16845 are the process IDs of each respective process. Thus, background task 1 is process ID 16843.

You can manipulate your background jobs from the command line using these labels. For instance, to terminate a command, type \texttt{kill \%N}, where \texttt{N} is the command's label. To move a command from the background to the foreground, type \texttt{fg \%N}:

\begin{verbatim}
$ sleep 10 & sleep 10 & sleep 10 &
[7] 17741
[8] 17742
[9] 17743
$ kill %7
$ jobs
[7] Terminated sleep 10
[8] Running sleep 10 &
[9] Running sleep 10 &
$ fg %8
sleep 10
\end{verbatim}

Running multiple commands simultaneously and asynchronously from the command line is a great way to juggle your own set of tasks. A long-running job -- say, number crunching or a large compilation -- is perfect to place in the background. To capture the output of each background command, consider redirecting the output to a file, using the redirection operators \texttt{>}, \texttt{>&}, \texttt{>>}, and \texttt{>>&}. Whenever a background command finishes, the shell prints an alert message before the next prompt:

\begin{verbatim}
$ whoami
mstreicher
[8] Done sleep 10
[9] Done sleep 10
$
\end{verbatim}

\textbf{To the great process pool in the sky}

Some processes live forever (such as init), and some processes reincarnate themselves into a new form (such as your shell). Ultimately, most processes die of natural causes -- a program runs to completion.

Additionally, you can place a process in a kind of suspended animation, where it waits to be reanimated. And as the previous example shows, you can terminate a process prematurely with \texttt{kill}.

If a command is running in the foreground and you want to suspend it, press \texttt{Control-Z}:

\begin{verbatim}
$ sleep 10
(Press Control-Z)
[1]+ Stopped sleep 10
$ ps
PID PPID USER COMMAND     S STIME     TIME
18195 16351 mstreic sleep 10 T 12:44 00:00:00
\end{verbatim}
The shell has suspended the command and assigned it a label for convenience. You can use this label as before to terminate the job or return it to the foreground. You can also use the `bg` command to resume the process in the background:

```
bg %1
[1]+ sleep 10 &
```

If a command is running in the foreground and you want to terminate it, press `Control-C`:

```
$ sleep 10
(Press Control-C
$ jobs
$
```

Your shell makes suspending and terminating a process easy, but a little voodoo is working beneath the shell's innocent facade. Internally, your shell uses UNIX signals to affect the state of processes. A signal is an event, and it's used to alert a process. The operating system originates many signals, but you can send signals from one process to another, or even have a process signal itself.

UNIX includes a wide variety of signals, most of which have a special purpose. For example, if you send signal `SIGSTOP` to a process, the process suspends. (For a complete list of signals, type `man 7 signal` or type `kill -L`). You send signals with the `kill` command:

```
$ sleep 20 &
[1] 19988
$ kill -SIGSTOP 19988
$ jobs
[1]+ Stopped sleep 20
```

Initially, the `sleep` command started in the background with process ID 19988. After sending `SIGSTOP`, the process changed state, becoming suspended or stopped. Sending another signal, `SIGCONT`, reanimates the process, and it resumes where it left off.

In other words, your shell sends `SIGSTOP` to the foreground process each time you press `Control-Z`. The `bg` command sends `SIGCONT`. And `Control-C` sends `SIGTERM`, which requests that the process terminate immediately.

Some signals can be blocked by a process, and applications can be designed to explicitly "catch" signals and react to each event in a special way. For instance, the system service `xinetd`, which launches other network services on demand, re-reads its configuration files upon the receipt of `SIGHUP`. On Linux, sending signals to `init` can change the system runlevel and even initiate system shutdown. (Here's a question: What's the difference between `kill %1` and `kill 1`?)

A process can even signal itself. Imagine that you're writing a game and want to give the user five seconds to respond. Your code can set a five-second timer and continue, say, redrawing the screen. When the timer runs out, `SIGALRM` is sent back to your process. Bzzzzt! Time's up!
(Here's the answer to the question: `kill %1` kills your background job labeled 1. `kill 1` terminates `init`, which is a signal to the operating system that it should shut down the entire machine.)

Still other signals are transmitted from the operating system to processes in special circumstances. A memory violation can spur `SIGSEGV`, killing the process instantly while leaving a core dump behind. One special signal, `SIGKILL`, can't be blocked or caught, and it kills a process immediately.

As with many other resources in UNIX, you can only signal processes that you own. This prevents you from terminating important system services and the processes of other users. The superuser, `root`, can signal any process.

**More magic demystified**

UNIX has many moving parts. It has system services, devices, memory managers, and more. Luckily, most of these complex machinations are hidden from view or are made convenient to use through user interfaces, such as the shell and windowing tools. Better yet, if you want to dive in, specialized tools, such as `top`, `ps`, and `kill`, all are readily available.

Now that you know how processes work, you can become your own one-person band. Just one request: *Freebird!*
Related topics

- Speaking UNIX: Check out other parts in this series.
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- Check out other articles and tutorials written by Martin Striecher:
  - Across developerWorks and IBM
- Search the AIX and UNIX library by topic:
  - System administration
  - Application development
  - Performance
  - Porting
  - Security
  - Tips
  - Tools and utilities
  - Java™ technology
  - Linux
  - Open source
- IBM trial software: Build your next development project with software for download directly from developerWorks.

© Copyright IBM Corporation 2007
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)