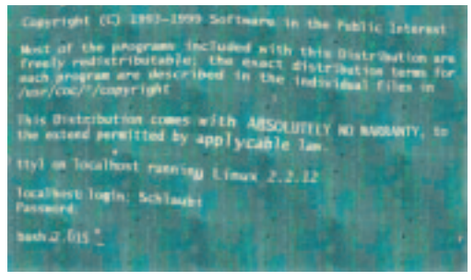
Take command

AN INTRODUCTION TO PROCESSES *PS, KILL* AND CONSORTS

BY HEIKE JURZIK



Even if lots of things can easily be controlled using graphical interfaces like KDE or GNOME, anyone wanting to get the most out of his Linux system cannot avoid using the command line. Apart from that, there are also many other situations where it's a good thing to know your way around the jungle of command lines.

Linux is capable of multitasking – in other words, a number of programs or processes can be executed by the processor at the same time. These processes are not actually dealt with simultaneously though; they are actually allocated a bit of computing time (a few fractions of a second) one after the other. The impression given however is that all the programs are running at the same time. This series is concerned with the management of processes and with tools which help you to do this.

Monitoring processes

You can find out which processes are running on your Linux system at any given moment using the command *ps*. If you type *ps x*, all processes currently running will be displayed:

```
chicken@asteroid:~$ ps x
  PID TTY      STAT   TIME COMMAND
  353 ?        SW     0:00 [.xsession]
  380 ?        S      0:00 /usr/bin/ssh-agent /ž
home/chicken/.xsession
...
  391 pts/1    SW     0:00 [bash]
  392 pts/0    S      0:00 -bash
...
16798 pts/5    S      0:00 ./zombie
16799 pts/5    Z      0:00 [zombie <defunct>]
...
16802 pts/0    R      0:00 ps x
```

Here you can see various pieces of information at a glance: the PID (Process Identification), the terminal in which the process was started, and the present status or condition of the program. The letters beneath *STAT* are fairly simple to understand: *S* means "sleeping", *R* means "running", *SW* means that the process is not only "asleep", but has also been moved out into the **SWap memory** . You can also tell this from the square brackets (*[ ]*).

Any process which is currently running must be in the RAM. If there isn't enough memory for the active processes, programs which are not currently running can be moved out into the Swap memory to make room for others. The obvious first choices for swapping out are the processes which are "sleeping" anyway and aren't actually doing anything.

In the list, you can also see another status flag: *Z*. This stands for "Zombie". A Zombie process is one which has stopped but has so far been unable to give back its return status to its **parent process** .

Apart from *S*, *R*, *SW* and *Z* there are some more letters which describe the current status. For example, if there is a *D* there, this is a process which

can no longer be "awoken" again. This type of thing can happen if the intention is to write into a file which is on a system mounted via NFS which has crashed in the meantime.

You can stop and restart your own processes yourself. Of course, you can't interfere with system processes, but if you want to stop a program, you can do it using *kill -STOP PID* (*kill -19 PID*). If the process is meant to continue to run, it is possible to convince it to do so using *kill -CONT PID* ("continue") (*kill -18 PID*) . A practical example: If, as *root* you wish to stop the *gpm* ("general purpose mouse" – mouse support on the console), you should first look for the process number (e.g. with *ps x* | *grep gpm*), then invoke *kill -STOP PID* . This process is now given a T as its status flag, which stands for "traced".

In the background, too – the shell as process manager

The shell receives inputs on the command line and decides whether these are internal commands or external programs. External programs are started as new processes. To the extent that these are dependent on keyboard inputs and screen outputs, they temporarily replace the shell in the terminal. For example, if one starts the program *ncftp*, this is in the foreground and receives the commands typed in. It is only when the program is ended that the shell comes into use again.

There are, of course, also programs which are not dependent on user inputs. If you want to carry on working in the shell, you can use "*program &* to" put the program into the background immediately. If the program is already running, the key combination [Ctrl+Z] and the following command *bg* (short for "background") have the same effect. With *fg* (short for "foreground"), it runs in the foreground again. Here's another trick: You can also put several programs into the background. If you then type in *jobs*, you will be given details of everything that's going on behind the scenes of the shell:

```
chicken@asteroid:~$ jobs
[1]-  Running    xkoules &
[2]+  Running    xskat &
```

Here you can see directly, as the first piece of information, the internal job ID (which is issued separately by each shell). This means the jobs can be managed, e.g. pushed individually into the foreground or background. Using the command *fg 1* you can for instance bring *xkoules* back to the front. This feature is a hangover from the days when you'd have to work at a single terminal. If there were two interactive programs, this was of course much easier to handle – you could for example put a text editor into the background while compiling the edited program, instead of shutting it down.

Not a forest, just one tree

The command *ps* obviously has any number of options with which you can control various selection options or properties. For example *ps au* also shows the processes of other users which are linked to some terminal or other. The *u* in this case ensures that additional details are displayed, including among other things the name of the process owner:

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root       307  0.0  1.0   1996   672 tty1     S    16:58   0:00 -bash
...
chicken   1427  0.0  1.5   2272  1000 pts/3    S    17:04   0:07 vim befehl
...
easter   16914  8.3  1.5   2628   988 pts/5    R    23:44   0:00 ps au
```

This also makes it clear what the current computing time percentage is (under *%CPU*), the proportion of memory (*%MEM*) or the time when the program starts (*START*); in older processes, instead of the time, the started date is displayed. With *ps r* only the processes running are displayed, and with *ps U username*, only those of a specified user. Another useful option is *ps l* (with "l" as in list), which enables you to find out something about the PPID ("parent process ID"). However a nicer overview of which process stems from which parent is delivered in a tree view, using the command *pstree*:

```
chicken@asteroid:~$ pstree
init-+-atd
     |-bash
     |-cron
     |-kdm-+-XF86_SVGA
     |     `-kdm---.xsession-+-fvwm2
     |                       |-ssh-agent
     |                       |-xclock
     |                       |-xeyes
...
```

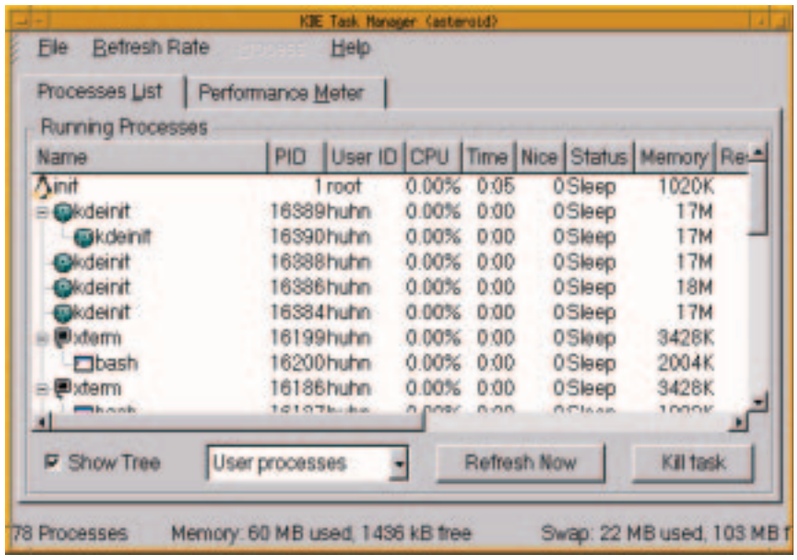
This type of overview can also be produced using *ktop* from KDE (see Fig. 1), if you have clicked on the option *show tree* .

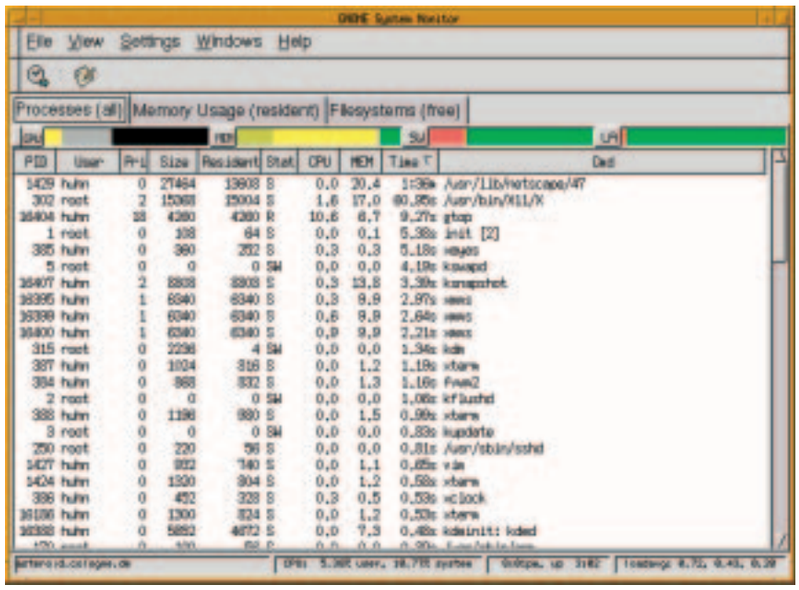
ps offers so many parameters that it is not possible to list them all here. Take a look at the man page and "knit" your own output format. The program *top* periodically conveys a useful selection. For GNOME, too, there is *top* with a graphical interface (see Fig. 2) with the practical additional function *Filesystems (free)*.

If you ever lose control over a process, you can terminate it by force. This is where the command *kill* comes into play. This can – as already mentioned – send all possible signals to processes. There are various signals for ending a process, which differ from each other as to whether they can be ignored or intercepted and processed by the process. With *kill -l* (with "l" as in list) you get an overview of all possible signals, including the ones used internally

Swap memory: *To enlarge the total amount of memory available, it is possible to supplement the physical working memory (RAM) with special areas on the hard disk called swap partitions or swap files. These are made by root with the command mkswap (see also the man page on this command).*

Parent process: *Every process, with the exception of the very first one (init), has been started by another. This other process is called the parent process, and the newly started one is called the child. If the child has been terminated, it gives back its status to the parent. If the latter has in the meantime been terminated itself, the reverse succession goes back to init.*





[top] Fig. 1: KDE's ktop offers a nice illustration of the tree structure

[above] Fig. 2: gtop is GNOME's front-end for process monitoring

by the system (e.g. *SIGSEGV* for the universally-popular memory overwriter):

```
chicken@asteroid:~$ kill -l
 1) SIGHUP     2) SIGINT   3) SIGQUIT  4) SIGILL
 5) SIGTRAP    6) SIGABRT  7) SIGBUS   8) SIGFPE
 9) SIGKILL   10) SIGUSR1 11) SIGSEGV 12) SIGUSR2
13) SIGPIPE   14) SIGALRM 15) SIGTERM 17) SIGCHLD
18) SIGCONT   19) SIGSTOP 20) SIGTSTP 21) SIGTTIN
22) SIGTTOU   23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR    31) SIGSYS
```

You've already met *SIGSTOP* and *SIGCONT* – and there are two more which are of practical interest.

- With *SIGTERM* (15) you can "ask" a program to terminate itself (it can still process the signal internally).
- With *SIGKILL* (9) on the other hand the program is unconditionally "shot down". This can be achieved with either *kill -9 PID* or with the long form *kill -KILL PID* .
- Also of interest is *kill -HUP PID* (*kill -1 PID*), to read in modifications in configuration files again.

For example a *kill -HUP* on the *inetd*-PID forces the Internet daemon to read in anew its configuration file */etc/inetd.conf*.

killall means you get them all...

As a rule you will not know the ID of a process which you want to terminate: You see something like a frozen Netscape window and would prefer to close this using *kill -9 PID* . Equipped only with the tools *ps* and *kill*, you must now first use

```
ps aux | grep netscape
```

to search for the process-ID of Netscape and then include this in the *kill* command – but there is a simpler way: a

```
killall -9 netscape
```

achieves precisely that (always assuming that the running program is also called *netscape* and not e.g. *netscape-communicator*). But be careful: This will actually terminate all matching processes; so for example *killall bash* shoots down all bash shells on the computer – including the one in which you have entered the *killall* command.

Nice try!

Now take another look at the output from *top* : Here the letters NI appear at one point – decrypted, this stands for "nice". With the command of the same name, processes can be assigned execution priority: Normal users can only reduce the priority, but *root* is allowed to increase it too. A program running at a lower priority is only then given computing time if this is not required by a process are started with a higher priority. By default, programs with the *nice* value 0, a value of -20 means top priority, +19 is the lowest. If no value is stated, *nice* starts programs with the value 10.

If you start *top* e.g. with *nice top*, you will see for this process:

```
  PID USER     PRI  NI  SIZE  RSS SHARE STAT ➥
LIB %CPU %MEM   TIME COMMAND
17827 chicken   19  10  1168 1168   704 R N  ➥
  0  2.1  1.8   0:01 top
```

If *top* is intended to run at the lowest priority, you take *nice -19 top* (for the highest level *nice --20 top*). If you don't like the double minus sign, you can also use the option "-n" and write for instance *nice -n 19 ...* or *nice -n -20 ...* .

The tools discussed here do of course have more options besides those mentioned. In particular *ps* can easily confuse you, because differing *ps* versions are in circulation, with different options. Should anything not function as expected, then – as usual – the best thing to do is look in the manual. ■