- Devin - http://www.devin.com.br -

Processos no Linux

Posted By Hugo Cisneiros (Eitch) On 24 de maio de 2009 @ 1:27 am In Linux, Tutoriais | 4 Comments

Saber o que está sendo executado na máquina é essencial para entender o funcionamento. Este tutorial traz uma introdução sobre os processos e threads no Linux: como listá-los, finalizá-los, interpretá-los.

Cada programa executado, desde a inicialização do sistema, é definido com o que chamamos de processo. Cada um desses processos recebe um número de identificação próprio, chamado PID (Process ID). Além do PID, cada processo tem um conjunto de informações como: nome do comando, uso da memória, usuário e grupo que o executou, entre outros.

As informações de todos os processos do sistema ficam armazenadas no pseudo-diretório /proc. Dentro deste diretório, cada sub-diretório numérico contém as informações do processo com o número PID correspondente. É deste lugar que os comandos relacionados aos processos retiram suas informações.

ps – Listar processos

```
Sintaxe: $ ps [opções]
```

Lista os processos em execução, apresentando o PID e outras informações sobre o processo, como o comando executado (CMD) e estado atual do processo (STAT).

Exemplo:

```
$ ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   1932   704 ?        Ss   Aug29   0:02 init [2]
root         2  0.0  0.0      0     0 ?        S<   Aug29   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S<   Aug29   0:00 [migration/0]
root         4  0.0  0.0      0     0 ?        S<   Aug29   0:06 [ksoftirqd/0]
...corte...
daemon    2998  0.0  0.0   1868   432 ?        Ss   Aug29   0:00 /usr/sbin/atd
root      3017  0.0  0.0   3272   928 ?        Ss   Aug29   0:00 /usr/sbin/cron
root      3074  0.0  0.0   2972   644 ?        Ss   Aug29   0:00 /usr/bin/kdm
root      3077  1.7 18.5 390652 384220 tty7    SLs+ Aug29  39:30 /usr/bin/X -br
...corte...
root      3092  0.0  0.0   1608   508 tty1     Ss+  Aug29   0:00 /sbin/getty
root      3093  0.0  0.0   1608   508 tty2     Ss+  Aug29   0:00 /sbin/getty
root      3094  0.0  0.0   1608   504 tty3     Ss+  Aug29   0:00 /sbin/getty
root      3095  0.0  0.0   1608   504 tty4     Ss+  Aug29   0:00 /sbin/getty
root      3096  0.0  0.0   1608   504 tty5     Ss+  Aug29   0:00 /sbin/getty
root      3097  0.0  0.0   1608   508 tty6     Ss+  Aug29   0:00 /sbin/getty
eitch  9403  4.5  0.1   5900  3364 tty1     S    23:50   0:00 -bash
eitch  9416  2.0  0.1   5304  2388 tty1     S+   23:50   0:00 vi testando
```

O parâmetro aux faz com que o comando ps mostre todos os processos do sistema, associado aos seus respectivos usuários e de forma detalhada. Nesta forma detalhada, podemos ver vários “campos” especificados na primeira linha. Eles são:

USER	Usuário que iniciou o processo (dono).
PID	Número único do processo.
%CPU	Utilização da CPU em porcentagem.
%MEM	Utilização da memória física do sistema em porcentagem.
VSZ	Memória virtual utilizada pelo processo, inclui a memória física e utilizada por bibliotecas compartilhadas.
RSS	Memória física utilizada pelo processo. Inclui memória utilizada por bibliotecas compartilhadas.
TTY	Terminal ao qual o processo pertence. Quando não há um terminal controlando (como no caso dos processos do sistema, kernel e processos do servidor gráfico) aparecerá o sinal de interrogação.
STAT	Estado atual do processo (mais detalhes depois).
START	A hora em que o processo foi iniciado. Caso a hora seja do dia anterior, é representado pelo dia e mês.
TIME	Tempo cumulativo de CPU, ou seja, por quanto tempo o processo utilizou a CPU.
COMMAND	O comando executado e todos seus argumentos. Caso o tamanho do comando seja maior do que a linha do terminal, ele ignora o resto (não passa para a próxima linha). Para mostrar todo o argumento, utilize o parâmetro w para ajustar o comprimento.

Em relação ao estado do processo, uma letra estará representando um destes estados:

D	Descansando enquanto espera por outra ação (geralmente E/S), sem possibilidade de interrupção.
R	Executando (Running).

S	Descansando enquanto espera por outra ação (esperando algum evento ser completado), com possibilidade de interrupção.
T	Parado, suspendido. Talvez pelo gerenciamento de tarefas da shell (CTRL+Z).
Z	Zumbi. SINISTRO! O processo foi terminado mas não foi removido por quem o chamou.

Uma outra forma de ver os processos, agora organizados por árvore:

```
$ ps axjf
```

top – Lista processos em tempo real

```
Sintaxe: $ top [opções]
```

Mostra ao usuário os processos ativos no sistema da mesma forma que o ps, mas em tempo real e em uma certa ordem. Por padrão, o top mostra nas primeiras linhas os processos que mais gastam processamento (em porcentagem).

Ao executar o top sem argumentos, sua taxa de atualização na tela é de 3 em 3 segundos. Para mudar para, por exemplo, 1 segundo:

```
$ top -d 1
```

Uma vez dentro do programa, a tecla h mostra a ajuda. Uma das opções úteis é a tecla f que permite especificar a ordem das linhas de acordo com os campos (mais utilização de cpu, memória, maior número UID de usuário, entre outros).

pstree – Mostra processos em forma de árvore hierárquica

```
Sintaxe: $ pstree [opções]
```

Mostra de forma simples e utilizando caracteres ASCII uma árvore hierarquica dos processos do sistema.

Sinais de Processos

Todo processo, além da entrada padrão que pode ser controlada dentro do programa, também pode receber o que chamamos de “sinal”. Este sinal é o modo que o sistema operacional tem para lidar com o processo. Um sinal pode dizer ao programa para terminar, ou outro sinal pode simplesmente terminar o programa sem dizer nada. Apertar um CTRL+C (Interrupção) no terminal é um sinal que o sistema manda para o programa atual.

Para saber a lista de sinais, digite:

```
$ man 7 signal
```

Quem faz uma aplicação pode programar para que quando um sinal for recebido pelo processo, o programa se comporte de uma certa maneira. O sinal SIGTERM (15) por exemplo, é chamado quando o programa é finalizado normalmente. Já o sinal SIGINT (2) é chamado quando o usuário aperta o CTRL+C e o programa tem que lidar com essa interrupção.

Os únicos sinais que são forçados pelo kernel são os sinais SIGKILL (9) e SIGSTOP (19). Neste caso, não importa o que o programador tentou fazer, vai ser executada as funções dos sinais nos processos.

Os sinais mais importantes são: SIGHUP (1), SIGINT (2), SIGKILL (9), SIGSEGV (11), SIGTERM (15) e SIGSTOP (19).

O SIGHUP (1) é geralmente utilizado pelos programas para recarregar seus arquivos de configuração. Um exemplo de programa que utiliza o SIGHUP para este fim é o inetd/xinetd.

O SIGINT (2) é recebido quando o usuário aperta a combinação de teclas CTRL+C. A grande maioria dos programas utiliza esse sinal para indicar uma interrupção do programa, para parar o que estiver fazendo e continuar com outra ação (ou finalizar o programa).

O SIGKILL (9) é o sinal que mata o processo. Não importa o que o processo esteja fazendo ou se ele é importante, o kernel irá forçar a sua finalização imediatamente. Este sinal não pode ser bloqueado ou rejeitado pelo programa.

O SIGSEGV (11) é um sinal recebido quando há alguma falha na alocação de memória. Também conhecido como “Segmentation Fault”, ele geralmente indica um problema no programa ou na pior das hipóteses problemas na memória física do computador.

O SIGTERM (15) é recebido pelos programas dizendo-os para finalizar de forma normal. É equivalente a fechar uma janela em um ambiente gráfico.

O SIGSTOP (19) é o sinal que suspende os programas, ou “os deixam imóveis” para poder manipulá-los como no uso do CTRL+Z na shell, ou então quando se está executando um trace em um programa.

kill, killall – Envia um sinal ao processo

```
Sintaxe: $ kill [-SINAL] <PID>
Sintaxe: $ killall [-SINAL] <nome do processo>
```

Apesar do nome, o comando kill não mata um processo e sim apenas manda um sinal para ele. Mas como já sabemos, há um sinal onde o kernel literalmente mata o processo. Para obter uma lista dos sinais suportados pelo sistema através do comando kill, digite:

```
$ kill -l
```

Para o comando kill, precisamos primeiro identificar o seu número PID, para depois mandar o sinal.

Por exemplo, temos um processo executando (vim) e queremos que ele seja morto (SIGKILL) sem aviso prévio:

```
$ ps aux
 USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
...corte...
eitch    2315  0.1  0.6   9364  3544 pts/1    S+   12:46   0:02 vim arquivo
```

Uma vez obtido o PID do processo:

```
$ kill -9 2315
```

Então o processo 2315 será morto sem piedade através do sinal SIGKILL.

Neste exemplo anterior, tivemos que obter o PID do processo para enviar um sinal. Com o comando killall, podemos mandar um sinal para todos os programas que têm um certo nome no comando. Utilizando desta forma, não é preciso saber o PID, mas por outro lado se houver dois processos com o mesmo nome, o sinal será mandado para os dois.

Matando o mesmo processo do exemplo anterior, agora utilizando o comando killall:

```
$ killall vim
```

Prioridades de Processos

Muita gente não sabe, mas nos sistemas Linux mais modernos, podemos ter um controle de prioridade de processos utilizando a CPU. Estas prioridades funcionam para favorecer melhor um comando à outro. Por exemplo:

- Programa 1 tem prioridade “alta”
- Programa 2 tem prioridade “normal”

Se o Programa 2 resolve gastar quase todo o processamento no sistema (como por exemplo: compactação de arquivos em bzip2), os programas que estão em prioridade alta não serão completamente afetados.

Se o Programa 1 estivesse em prioridade “normal”, ele teria que dividir o processamento com o Programa 2, mesmo que ele usasse pouquinho. Mas como ele está em “alta” prioridade, ele sempre vai ter a preferência no sistema, ao invés do Programa 2.

A faixa de prioridades é:

- -20 (Prioridade Máxima)
- -19
- ...
- -1
- 0 (Prioridade Padrão)
- 1
- ...
- 18
- 19 (Prioridade Mínima)

Sendo assim, por padrão, todo comando executado “normalmente” recebe a prioridade 0 (Zero). E quanto menor for esse número, maior a prioridade do processo na CPU (Sim, é o contrário! Quanto menor, maior; quanto maior, menor! :).

Por padrão, os comandos ligados ao kernel e o sistema operacional em si têm prioridade inferior a 0 (aqui no meu sistema, vejo vários processos com prioridade -5). Isso nos faz lembrar que **você precisa ser root para configurar um processo com prioridade menor que 0**. Isso faz sentido, já que em um sistema multi-usuário, os usuários normais compartilham o padrão 0, e podem escolher apenas se o processo pode ter uma prioridade mais baixa, assim não interferindo com os processos do sistema, que teoricamente são mais importantes.

Para listar as prioridades dos processos, podemos utilizar o ps:

```
$ ps axo user,ni,command
USER      NI COMMAND
root       0 init [2]
root      -5 [kthreadd]
root       - [migration/0]
root      -5 [ksoftirqd/0]
root       - [watchdog/0]
root       - [migration/1]
[...]
root       0 /usr/sbin/cron
root       0 /usr/bin/kdm -config /var/run/kdm/kdmrc
root       0 /usr/bin/X -br -nolisten tcp :0 vt7 -auth /var/run/xauth/A:0-QPhClu
root       0 /sbin/getty 38400 tty1
[...]
hugo       0 kdeinit Running...
hugo       0 dcopserver [kdeinit] --nosid
hugo       0 klauncher [kdeinit] --new-startup
hugo       0 kded [kdeinit] --new-startup
hugo       0 kwrapper ksmserver
[..]
root      -2 dhclient3 -pf /var/run/dhclient.eth0.pid -lf /var/lib/dhcp3/dhclient.eth0.leases eth0
hugo       0 keepassx
hugo       0 /bin/bash
```

```
hugo      19 tail -f teste.txt
hugo       0 ps axo user,ni,command
```

Neste comando, a primeira coluna é o usuário que está executando o processo; a segunda é a prioridade de execução; e a terceira é o comando em si. Dá pra perceber bem que a maioria dos comandos recebe 0 como prioridade, com algumas excessões.

nice, renice – Configura prioridades de CPU de um comando ou processo

```
Sintaxe: $ nice [-n PRIORIDADE] [comando ...]
Sintaxe: $ renice <PRIORIDADE> [-p PID]
```

Uma vez que já sabemos como funciona a prioridade de processos no Linux, podemos controlar estas prioridades com os comandos *nice* e *renice*.

Por exemplo, supomos que eu esteja trabalhando em um servidor Web bastante movimentado. O Apache (servidor HTTP) está sendo executado com prioridade 0 (normal), e está muito ocupado servindo todas as páginas dinâmicas para os vários usuários ao mesmo tempo. Então, toca o telefone e me pedem para fazer um backup de todas as páginas do servidor, algo em torno de uns 10gb, tudo localizado no diretório /var/www.

Para fazer o backup, eu utilizei o seguinte comando:

```
tar -zcf /var/lib/backup/backup-htdocs-20090520.tar.gz /var/www
```

Ou seja, agrupar e compactar todo o diretório /var/www e seu conteúdo no arquivo backup-htdocs-20090520.tar.gz. Estaria certo eu fazer assim?

A resposta é sim e não. Com o comando acima, seria criado com sucesso o backup que precisava, mas o custo é muito alto... Enquanto o sistema compacta todos os 10gb e colocava no arquivo, o Apache tinha que **dividir** as tarefas dele com esse comando! Isso significa que os usuários Web teriam suas páginas entregues de forma mais lenta... Como a prioridade aqui é servidor as páginas, teríamos que usar o comando:

```
nice -n 19 tar -zcf /var/lib/backup/backup-htdocs-20090520.tar.gz /var/www
```

Neste caso, o backup seria feito do mesmo jeito, mas a prioridade dele seria a mínima! Toda vez que o Apache precisasse de muito processamento, o sistema ia dar prioridade total ao Apache, deixando um pouco de lado o backup. Resultado: os usuários do Apache não seriam de maneira alguma afetados e o sistema não iria ter uma carga maior do que o esperado. Ao invés disso, o backup iria demorar mais, pois tinha que esperar primeiro os outros processos usarem a CPU, mas seria feito da mesma forma.

Então é bem simples: primeiro escolhemos a prioridade antes de executar o comando, e fazemos o mesmo comando, colocando antes dele o "*nice -n*". Mas e se o comando já está sendo executado, como no primeiro exemplo aqui? Neste caso, usamos o comando *renice* para redefinir a prioridade.

Para utilizar o comando renice, precisamos achar primeiro o PID do processo, depois executar o comando neste PID, exemplo:

```
$ ps axo user,pid,ni,command | grep tar
hugo      4221   0 tar -zcf /var/lib/backup/backup-htdocs-20090520.tar.gz /var/www

$ renice 19 -p 4221
4221: old priority 0, new priority 19

$ ps axo user,pid,ni,command | grep tar
hugo      4221  19 tar -zcf /var/lib/backup/backup-htdocs-20090520.tar.gz /var/www
```

Nos comandos acima, eu: achei o processo que estava executando o backup, e identifiquei que o PID dele era 4221. Note também que a prioridade do processo é 0 (normal). Como queremos colocar o comando em uma prioridade baixa, então utilizei o comando **renice** para redefinir essa prioridade: de 0 para 19. Depois utilizei novamente o comando *ps* para listar o processo e vi que a prioridade dele tinha mudado.

Como último exemplo, lembre-se que usuários comuns (aqui em nosso caso, o usuário hugo) não podem ultrapassar prioridades maiores que 0, ou seja, não podem colocar prioridades negativas. Veja o que acontece no mesmo exemplo anterior:

```
$ nice -n -10 tar -zcf /var/lib/backup/backup-htdocs-20090520.tar.gz /var/www
nice: cannot set niceness: Permission denied
```

Quando tentei utilizar a prioridade -10, o sistema não deixou e retornou a mensagem de permissão negada. Se fôssemos root, neste caso, iria funcionar.

lsof – Listar arquivos abertos

```
Sintaxe: $ lsof [opções] [arquivo]
```

O comando lsof é um dos mais importantes comandos para quem administra sistemas Linux, principalmente na área de segurança. Este comando lista **todos** os arquivos abertos por todos os processos. Aqui, quando eu falo arquivo, não são apenas arquivos comuns, mas sim recursos que funcionam como arquivos (podem ser abertos, mapeados na memória, entre outros). Isso inclui bibliotecas, sockets, arquivos comuns, diretórios e por aí vai.

Em outras palavras, este comando nos fornece um mapeamento completo do que o programa está usando no sistema. Lembre-se que usando apenas o comando lsof, esta lista fica muito grande, pois mostra todos os arquivos de todos os processos. Por exemplo:

```
$ lsof -n
[...]
```

```
bash      4409          hugo  cwd       DIR      254,1     4096 2752514 /home/hugo
bash      4409          hugo  rtd       DIR      254,1     4096       2 /
bash      4409          hugo  txt       REG      254,1   700492 5849112 /bin/bash
bash      4409          hugo  mem       REG      254,1    42504 5652815 /lib/i686/cmov/libnss_files-2.7.so
bash      4409          hugo  mem       REG      254,1    38444 5652817 /lib/i686/cmov/libnss_nis-2.7.so
bash      4409          hugo  mem       REG      254,1    87800 5652810 /lib/i686/cmov/libnsl-2.7.so
bash      4409          hugo  mem       REG      254,1    30436 5652811 /lib/i686/cmov/libnss_compat-2.7.so
bash      4409          hugo  mem       REG      254,1  1282816 5213150 /usr/lib/locale/locale-archive
bash      4409          hugo  mem       REG      254,1  1413540 5652651 /lib/i686/cmov/libc-2.7.so
bash      4409          hugo  mem       REG      254,1     9680 5652657 /lib/i686/cmov/libdl-2.7.so
bash      4409          hugo  mem       REG      254,1   202188 7406484 /lib/libncurses.so.5.6
bash      4409          hugo  mem       REG      254,1    25700 3178882 /usr/lib/gconv/gconv-modules.cache
bash      4409          hugo  mem       REG      254,1   113248 5653602 /lib/ld-2.7.so
bash      4409          hugo    0u      CHR      136,3         5 /dev/pts/3
bash      4409          hugo    1u      CHR      136,3         5 /dev/pts/3
bash      4409          hugo    2u      CHR      136,3         5 /dev/pts/3
bash      4409          hugo  255u      CHR      136,3         5 /dev/pts/3
[...]
```

No exemplo acima, eu peguei apenas um fragmento do comando, indicando o que o comando *bash* está fazendo. Dá pra ver que bibliotecas ele está usando, onde ele está atuando, entre outros. O parâmetro "-n", que usei no exemplo acima, serve para que se o comando retornar algum endereço de rede (IP, por exemplo), ele não tente resolver com DNS, assim o retorno do comando fica mais rápido.

Alguns dos usos mais comuns incluem:

- Ver se algum processo está escutando uma porta na rede suspeita, ou conectado em algum lugar suspeito. Por exemplo, vários scripts de invasão ficam escondidos no sistema (com nomes de outros processos), conectados a servidores de IRC desconhecidos. Com o lsof, dá pra saber que estes comando estão fazendo algo que não é bem o que deveriam fazer ;) ;
- Ver que processo está usando um certo arquivo (lsof);
- Ver exatamente que tipos de conexão estão sendo feitas no sistema;
- Medir as memórias utilizadas pelos processos.

Por exemplo, quero ver todos os processos que utilizam o arquivo /dev/null:

```
$ lsof /dev/null
COMMAND    PID USER   FD   TYPE DEVICE SIZE NODE NAME
x-session 3193 hugo    0r   CHR    1,3       617 /dev/null
dbus-laun 3228 hugo    0r   CHR    1,3       617 /dev/null
dbus-laun 3228 hugo    1u   CHR    1,3       617 /dev/null
dbus-laun 3228 hugo    2u   CHR    1,3       617 /dev/null
dbus-laun 3228 hugo    4u   CHR    1,3       617 /dev/null
dbus-daem 3229 hugo    0u   CHR    1,3       617 /dev/null
dbus-daem 3229 hugo    1u   CHR    1,3       617 /dev/null
dbus-daem 3229 hugo    2u   CHR    1,3       617 /dev/null
dbus-daem 3229 hugo    4u   CHR    1,3       617 /dev/null
kwrapper  3275 hugo    0r   CHR    1,3       617 /dev/null
gconfd-2  3393 hugo    0u   CHR    1,3       617 /dev/null
gconfd-2  3393 hugo    1u   CHR    1,3       617 /dev/null
gconfd-2  3393 hugo    2u   CHR    1,3       617 /dev/null
gconfd-2  3393 hugo    3u   CHR    1,3       617 /dev/null
```

Este comando é bem parecido com o comando "fuser /dev/null", que também mostra que processos estão utilizando o arquivo /dev/null, mas o lsof nos dá mais detalhes.

Quero ver agora que processos estão utilizando conexões TCP no meu sistema:

```
$ lsof -n | grep TCP
firefox-b 3327          hugo   12u     IPv4      41100             TCP 192.168.1.100:58945->66.102.1.100:www (ESTABLISHED)
firefox-b 3327          hugo   39u     IPv4      41951             TCP 192.168.1.100:45640->72.14.247.18:https (ESTABLISHED)
firefox-b 3327          hugo   42u     IPv4      39570             TCP 192.168.1.100:47900->72.14.247.19:https (ESTABLISHED)
wish8.5   3535          hugo    7u     IPv4       9331             TCP 127.0.0.1:65182 (LISTEN)
wish8.5   3535          hugo    8u     IPv4       9578             TCP 192.168.1.100:46238->207.46.106.108:msnp (ESTABLISHED
```

Repare que na penúltima linha do comando anterior, o processo "wish8.5" está escutando (LISTEN) uma conexão TCP na porta 65182.
Que tal a gente saber todos os processos que estão escutando portas de rede?

```
# lsof -n | grep LISTEN
rpcbind    1264       rpc    8u     IPv4       4159                TCP *:sunrpc (LISTEN)
snmpd      1341      root    8u     IPv4       4561                TCP 127.0.0.1:smux (LISTEN)
sshd       1351      root    3u     IPv4       4451                TCP *:domain (LISTEN)
postmaste  1553  postgres    3u     IPv6       4895                TCP *:postgres (LISTEN)
postmaste  1553  postgres    4u     IPv4       4896                TCP *:postgres (LISTEN)
proftpd    1601       ftp    0u     IPv6       4858                TCP *:ftp (LISTEN)
mysqld     7882     mysql   10u     IPv4    3835186                TCP *:mysql (LISTEN)
named     31990     named   21u     IPv4  107763358                TCP 127.0.0.1:domain (LISTEN)
named     31990     named   60u     IPv4  107763397                TCP 127.0.0.1:rndc (LISTEN)
```

Repare que para o comando acima, eu utilizei o **root** (por isso o # antes do comando). Isto é necessário pois quando utilizamos o lsof como usuário normal, não temos todas as permissões necessárias para verificar todos os processos, então a saída do comando vai ficar bem restrita. Executando como root, podemos ter todas as informações possíveis.
