103.4 Fluxos, pipes (canalização) e redirecionamentos de saída

Referência ao LPI objectivo

LPIC-1 v5, Exam 101, Objective 103.4

Peso

4

Áreas chave de conhecimento

- Redirecionamento da entrada padrão, da saída padrão e dos erros padrão.
- Canalização (piping) da saída de um comando à entrada de outro comando.
- Usar a saída de um comando como argumento para outro comando.
- Enviar a saída de um comando simultaneamente para a saída padrão e um arquivo.

Segue uma lista parcial dos arquivos, termos e utilitários utilizados

- `tee`
- `xargs`

103.4 Lição 1

Certificação:	LPIC-1
Versão:	5.0
Tópico:	103 Comandos GNU e Unix
Objetivo:	103.4 Usando fluxos, pipes e redirecionamentos
Lição:	1 de 2

Introdução

Todos os programas de computador seguem o mesmo princípio geral: os dados recebidos de alguma fonte são transformados para gerar um resultado inteligível. No contexto do shell do Linux, a fonte de dados pode ser um arquivo local, um arquivo remoto, um dispositivo (como um teclado), etc. A saída do programa geralmente é exibida em uma tela, mas também é comum armazenar os dados de saída em um sistema de arquivos local, enviar para um dispositivo remoto, reproduzi-lo em alto-falantes de áudio, etc.

Os sistemas operacionais inspirados no Unix, como o Linux, oferecem uma grande variedade de métodos de entrada/saída. Em particular, o método dos *descritores de arquivo* permite associar dinamicamente números inteiros a canais de dados, para que um processo possa referenciá-los como seus fluxos de dados de entrada/saída.

Os processos padrão do Linux têm três canais de comunicação abertos por padrão: o canal de *entrada padrão* (na maioria das vezes simplesmente chamado de *stdin*), o canal de *saída padrão* (*stdout*) e o canal de *erro padrão* (*stderr*). Os descritores numéricos de arquivo atribuídos a esses canais são `0` para stdin, `1` para stdout e `2` para stderr. Os canais de comunicação também são acessíveis por meio dos dispositivos especiais `/dev/stdin`, `/dev/stdout` e `/dev/stderr`.

Esses três canais de comunicação permitem que os programadores escrevam códigos que lêem e gravam dados sem se preocupar com o tipo de mídia de onde vêm ou para o qual vão. Por exemplo, se um programa precisa de um conjunto de dados como entrada, pode simplesmente solicitar os dados da entrada padrão; será fornecido aquilo que estiver sendo usado como entrada padrão. Da mesma forma, o método mais simples que um programa pode usar para exibir sua saída é escrevê-la na saída padrão. Em uma sessão comum do shell, o teclado é definido como stdin e a tela do monitor como stdout e stderr.

O shell Bash tem a capacidade de reatribuir os canais de comunicação ao carregar um programa. Ele permite, por exemplo, substituir a tela como a saída padrão e usar um arquivo no sistema de arquivos local como stdout.

Redirecionamentos

A reatribuição do descritor de arquivo de um canal no ambiente shell é chamada de *redirecionamento*. Um redirecionamento é definido por um caractere especial na linha de comando. Por exemplo, para redirecionar a saída padrão de um processo para um arquivo, o símbolo de *maior que* `>` é posicionado no final do comando e seguido pelo caminho até o arquivo que receberá a saída redirecionada:

```
$ cat /proc/cpuinfo >/tmp/cpu.txt
```

Por padrão, apenas o conteúdo que chega a stdout é redirecionado. Isso ocorre porque o valor numérico do descritor de arquivo deve ser especificado logo antes do símbolo de maior que e, quando não especificado, o Bash redireciona a saída padrão. Portanto, usar `>` é equivalente a usar `1>` (o valor do descritor de arquivo de stdout é `1`).

Para capturar o conteúdo de stderr, o redirecionamento `2>` deve ser usado. A maioria dos programas de linha de comando enviam informações de depuração e mensagens de erro para o canal de erro padrão. É possível, por exemplo, capturar a mensagem de erro gerada por uma tentativa de leitura de um arquivo inexistente:

```
$ cat /proc/cpu_info 2>/tmp/error.txt
$ cat /tmp/error.txt
cat: /proc/cpu_info: No such file or directory
```

Tanto stdout quanto stderr são redirecionados para o mesmo destino com `&>` ou `>&`. É importante não colocar nenhum espaço ao lado do "e" comercial, caso contrário o Bash o interpretará como uma instrução para executar o processo em segundo plano e não para executar o redirecionamento.

O destino deve ser um caminho para um arquivo gravável, como `/tmp/cpu.txt`, ou um descritor de

arquivo gravável. O destino de um descritor de arquivo é representado por um "e" comercial seguido pelo valor numérico do descritor de arquivo. Por exemplo, `1>&2` redireciona stdout para stderr. Para fazer o oposto, stderr para stdout, devemos usar `2>&1`.

Embora não seja muito útil, visto que existe uma maneira mais curta de executar a mesma tarefa, é possível redirecionar stderr para stdout e, em seguida, redirecioná-lo para um arquivo. Por exemplo, um redirecionamento para gravar stderr e stdout em um arquivo chamado `log.txt` pode ser escrito como `>log.txt 2>&1`. No entanto, o principal motivo para redirecionar stderr para stdout é permitir a análise de mensagens de depuração e erro. É possível redirecionar a saída padrão de um programa para a entrada padrão de outro programa, mas não é possível redirecionar diretamente o erro padrão para a entrada padrão de outro programa. Assim, as mensagens do programa enviadas para stderr primeiro precisam ser redirecionadas para stdout a fim de serem lidas pelo stdin de outro programa.

Para simplesmente descartar a saída de um comando, seu conteúdo pode ser redirecionado para o arquivo especial `/dev/null`. Por exemplo, `>log.txt 2>/dev/null` salva o conteúdo de stdout no arquivo `log.txt` e descarta o stderr. O arquivo `/dev/null` pode ser escrito por qualquer usuário, mas nenhum dado pode ser recuperado dele, pois não é armazenado em lugar nenhum.

Uma mensagem de erro é apresentada se o destino especificado não for gravável (se o caminho apontar para um diretório ou um arquivo somente leitura) e nenhuma modificação for feita no destino. No entanto, um redirecionamento de saída sobrescreve um destino gravável existente sem pedir nenhuma confirmação. Os arquivos são substituídos pelos redirecionamentos de saída, a menos que a opção `noclobber` esteja habilitada no Bash, o que pode ser feito para a sessão atual com o comando `set –o noclobber` ou `set –C`:

```
$ set -o noclobber
$ cat /proc/cpu_info 2>/tmp/error.txt
-bash: /tmp/error.txt: cannot overwrite existing file
```

Para remover a opção `noclobber` da sessão atual, execute `set +o noclobber` ou `set +C`. Para tornar a opção `noclobber` persistente, ela deve ser incluída no perfil Bash do usuário ou no perfil de todo o sistema.

Mesmo com a opção `noclobber` habilitada, é possível anexar dados redirecionados ao conteúdo existente. Usamos para isso um redirecionamento escrito com dois símbolos de maior que, `>>`:

```
$ cat /proc/cpu_info 2>>/tmp/error.txt
$ cat /tmp/error.txt
cat: /proc/cpu_info: No such file or directory
cat: /proc/cpu_info: No such file or directory
```

No exemplo anterior, a nova mensagem de erro foi anexada à existente no arquivo `/tmp/error.txt`. Se o arquivo ainda não existir, ele será criado com os novos dados.

A fonte de dados da entrada padrão de um processo também pode ser reatribuída. O símbolo de menor que `<` é usado para redirecionar o conteúdo de um arquivo para o stdin de um processo. Nesse caso, os dados fluem da direita para a esquerda: o descritor reatribuído é considerado como sendo 0 à esquerda do símbolo de menor que e a fonte de dados (um caminho para um arquivo) deve estar à direita do símbolo de menor que. O comando `uniq`, como a maioria dos utilitários de linha de comando para processamento de texto, aceita os dados enviados para stdin por padrão:

```
$ uniq -c </tmp/error.txt
      2 cat: /proc/cpu_info: No such file or directory
```

A opção `-c` faz com que o `uniq` exiba quantas vezes uma linha repetida aparece no texto. Como o valor numérico do descritor de arquivo redirecionado foi suprimido, o comando de exemplo é equivalente a `uniq -c 0</tmp/error.txt`. O uso de um descritor de arquivo diferente de `0` em um redirecionamento de entrada só faz sentido em determinados contextos, porque um programa pode possivelmente solicitar dados dos descritores `3`, `4`, etc. De fato, os programas podem usar qualquer número inteiro maior que 2 como novos descritores de arquivo para entrada/saída de dados. Por exemplo, o código em C a seguir lê dados do descritor de arquivo `3` e simplesmente os reproduz para o descritor `4`:

NOTE O programa deve gerenciar corretamente esses descritores de arquivo, caso contrário ele pode tentar uma operação inválida de leitura ou gravação e travar.

```
#include <stdio.h>

int main(int argc, char **argv){
  FILE *fd_3, *fd_4;
  // Open file descriptor 3
  fd_3 = fdopen(3, "r");
  // Open file descriptor 4
  fd_4 = fdopen(4, "w");
  // Read from file descriptor 3
  char buf[32];
  while ( fgets(buf, 32, fd_3) != NULL ){
    // Write to file descriptor 4
    fprintf(fd_4, "%s", buf);
  }
  // Close both file descriptors
  fclose(fd_3);
```

```
  fclose(fd_4);
}
```

Para testá-lo, salve o código de amostra como `fd.c` e compile-o com `gcc -o fd fd.c`. Este programa precisa que os descritores de arquivo 3 e 4 estejam disponíveis para poder ler e gravar neles. Como exemplo, o arquivo `/tmp/error.txt` criado anteriormente pode ser usado como fonte para o descritor de arquivo `3` e o descritor de arquivo `4` pode ser redirecionado para stdout:

```
$ ./fd 3</tmp/error.txt 4>&1
cat: /proc/cpu_info: No such file or directory
cat: /proc/cpu_info: No such file or directory
```

Do ponto de vista do programador, o uso de descritores de arquivo evita a obrigação de lidar com a análise de opções (parsing) e com os caminhos do sistema de arquivos. É possível até usar o mesmo descritor de arquivo como entrada e saída. Nesse caso, o descritor de arquivo é definido na linha de comando com os símbolos menor que e maior que, como em `3<>/tmp/error.txt`.

Here Document e Here String

Outra forma de redirecionar a entrada envolve os métodos *Here document* e *Here string*. O redirecionamento Here document permite digitar um texto de várias linhas que será usado como o conteúdo redirecionado. Dois símbolos de menor que `<<` indicam um redirecionamento de Here document:

```
$ wc -c <<EOF
> How many characters
> in this Here document?
> EOF
43
```

À direita dos dois símbolos de menor que `<<` está o termo de fim `EOF`. O modo de inserção termina assim que for inserida uma linha contendo apenas o termo de fim. Qualquer outro termo pode ser usado como termo de fim, mas é importante não colocar caracteres em branco entre o símbolo de menor que e o termo de fim. No exemplo acima, as duas linhas de texto foram enviadas para o stdin do comando `wc -c`, que exibe a contagem de caracteres. Assim como acontece com os redirecionamentos de entrada para arquivos, o stdin (descritor de arquivo `0`) é pressuposto se o descritor de arquivo redirecionado for suprimido.

O método de Here string é muito parecido com o método de Here document, mas para uma linha apenas:

```
$ wc -c <<<"How many characters in this Here string?"
41
```

Neste exemplo, a string à direita dos três sinais de menor é enviada para o stdin de `wc -c`, que conta o número de caracteres. As strings contendo espaços devem estar entre aspas, senão apenas a primeira palavra será usada como Here string e as restantes serão passadas como argumentos para o comando.

Exercícios Guiados

1. Além dos arquivos de texto, o comando `cat` também pode trabalhar com dados binários, como enviar o conteúdo de um dispositivo de bloco para um arquivo. Usando redirecionamento, como o `cat` pode enviar o conteúdo do dispositivo `/dev/sdc` para o arquivo `sdc.img` no diretório atual?

2. Qual é o nome do canal padrão redirecionado pelo comando `date 1> now.txt`?

3. Ao tentar sobrescrever um arquivo usando redirecionamento, um usuário recebe uma mensagem de erro informando que a opção `noclobber` está habilitada. Como essa opção pode ser desativada para a sessão atual?

4. Qual será o resultado do comando `cat <<.>/dev/stdout`?

Exercícios Exploratórios

1. O comando `cat /proc/cpu_info` exibe uma mensagem de erro porque `/proc/cpu_info` não existe. Para onde o comando `cat /proc/cpu_info 2>1` redireciona a mensagem de erro?

2. Ainda será possível descartar o conteúdo enviado para `/dev/null` se a opção `noclobber` estiver habilitada para a sessão de shell atual?

3. Sem usar `echo`, como o conteúdo da variável `$USER` poderia ser redirecionado para o stdin do comando `sha1sum`?

4. O kernel do Linux mantém links simbólicos em `/proc/PID/fd/` para cada arquivo aberto por um processo, onde *PID* é o número de identificação do processo correspondente. Como o administrador do sistema poderia usar esse diretório para verificar a localização dos arquivos de log abertos pelo `nginx`, supondo que seu PID é `1234`?

5. É possível fazer cálculos aritméticos usando apenas comandos internos do shell, mas cálculos de ponto flutuante requerem programas específicos, como o `bc` (*basic calculator*). Com o `bc` é possível até mesmo especificar o número de casas decimais, com o parâmetro `escala`. No entanto, o `bc` aceita operações apenas por meio de sua entrada padrão, geralmente inserida no modo interativo. Usando uma Here string, como a operação de ponto flutuante `scale=6; 1/3` pode ser enviada para a entrada padrão de `bc`?

Resumo

Esta lição cobre métodos para executar um programa redirecionando seus canais de comunicação padrão. Os processos do Linux usam esses canais padrão como *descritores de arquivo* genéricos para ler e gravar dados, tornando possível transferi-los arbitrariamente para arquivos ou dispositivos. A lição demonstra as seguintes etapas:

- O que são descritores de arquivos e qual seu papel no Linux.
- Os canais de comunicação padrão em todos os processos: *stdin*, *stdout* e *stderr*.
- Como executar corretamente um comando usando redirecionamento de dados, tanto na entrada quanto na saída.
- Como usar *Here Documents* e *Here Strings* nos redirecionamentos de entrada.

Os comandos e procedimentos abordados foram:

- Operadores de redirecionamento: `>`, `<`, `>>`, `<<`, `<<<`.
- Comandos `cat`, `set`, `uniq` e `wc`.

Respostas aos Exercícios Guiados

1. Além dos arquivos de texto, o comando `cat` também pode trabalhar com dados binários, como enviar o conteúdo de um dispositivo de bloco para um arquivo. Usando redirecionamento, como o `cat` pode enviar o conteúdo do dispositivo `/dev/sdc` para o arquivo `sdc.img` no diretório atual?

   ```
   $ cat /dev/sdc > sdc.img
   ```

2. Qual é o nome do canal padrão redirecionado pelo comando `date 1> now.txt`?

   Standard output ou stdout

3. Ao tentar sobrescrever um arquivo usando redirecionamento, um usuário recebe uma mensagem de erro informando que a opção `noclobber` está habilitada. Como essa opção pode ser desativada para a sessão atual?

   `set +C` ou `set +o noclobber`

4. Qual será o resultado do comando `cat <<./dev/stdout`?

   O Bash entrará no modo de entrada Heredoc e sairá quando um ponto final aparecer sozinho em uma linha. O texto digitado será redirecionado para stdout (impresso na tela).

Respostas aos Exercícios Exploratórios

1. O comando `cat /proc/cpu_info` exibe uma mensagem de erro porque `/proc/cpu_info` não existe. Para onde o comando `cat /proc/cpu_info 2>1` redireciona a mensagem de erro?

   Para um arquivo chamado `1` no diretório atual.

2. Ainda será possível descartar o conteúdo enviado para `/dev/null` se a opção `noclobber` estiver habilitada para a sessão de shell atual?

   Sim. `/dev/null` é um arquivo especial, não afetado por `noclobber`.

3. Sem usar `echo`, como o conteúdo da variável `$USER` poderia ser redirecionado para o stdin do comando `sha1sum`?

   ```
   $ sha1sum <<<$USER
   ```

4. O kernel do Linux mantém links simbólicos em `/proc/PID/fd/` para cada arquivo aberto por um processo, onde *PID* é o número de identificação do processo correspondente. Como o administrador do sistema poderia usar esse diretório para verificar a localização dos arquivos de log abertos pelo `nginx`, supondo que seu PID é `1234`?

   Emitindo o comando `ls -l /proc/1234/fd`, que exibirá os destinos de cada link simbólico no diretório.

5. É possível fazer cálculos aritméticos usando apenas comandos internos do shell, mas cálculos de ponto flutuante requerem programas específicos, como o `bc` (*basic calculator*). Com o `bc` é possível até mesmo especificar o número de casas decimais, com o parâmetro `escala`. No entanto, o `bc` aceita operações apenas por meio de sua entrada padrão, geralmente inserida no modo interativo. Usando uma Here string, como a operação de ponto flutuante `scale=6; 1/3` pode ser enviada para a entrada padrão de `bc`?

   ```
   $ bc <<<"scale=6; 1/3"
   ```

103.4 Lição 2

Certificação:	LPIC-1
Versão:	5.0
Tópico:	103 Comandos GNU e Unix
Objetivo:	103.4 Usando fluxos, pipes e redirecionamentos
Lição:	2 de 2

Introdução

Um aspecto da filosofia Unix afirma que cada programa precisa ter um propósito específico e não deve tentar incorporar recursos fora de seu escopo. Mas manter as coisas simples não significa que os resultados serão menos elaborados, já que diferentes programas podem ser encadeados para produzir uma saída combinada. O caractere de barra vertical `|`, também conhecido como símbolo *pipe*, pode ser usado para criar uma canalização (pipeline) conectando a saída de um programa diretamente à entrada de outro programa, ao passo que a *substituição de comandos* permite armazenar a saída de um programa em uma variável ou usá-lo diretamente como argumento para outro comando.

Pipes

Ao contrário dos redirecionamentos, com os pipes os dados fluem da esquerda para a direita na linha de comando e o destino é outro processo, não um caminho do sistema de arquivos, descritor de arquivo ou Here document. O caractere de barra vertical `|` manda o shell iniciar todos os comandos distintos ao mesmo tempo e conectar a saída do comando anterior à entrada do comando seguinte, da esquerda para a direita. Por exemplo, em vez de usar redirecionamentos, o conteúdo do arquivo `/proc/cpuinfo` enviado para a saída padrão por `cat` pode ser canalizado para o stdin de `wc` com o

seguinte comando:

```
$ cat /proc/cpuinfo | wc
   208    1184    6096
```

Na ausência do caminho para um arquivo, `wc` conta o número de linhas, palavras e caracteres que recebe em seu stdin, como é o caso no exemplo. Muitos pipes podem estar presentes em um comando composto. No exemplo a seguir, dois pipes são usados:

```
$ cat /proc/cpuinfo | grep 'model name' | uniq
model name      : Intel(R) Xeon(R) CPU           X5355  @ 2.66GHz
```

O conteúdo do arquivo `/proc/cpuinfo` produzido por `cat /proc/cpuinfo` foi canalizado para o comando `grep 'model name'`, que em seguida seleciona apenas as linhas contendo o termo `model name`. A máquina que executa o exemplo tem muitas CPUs, portanto existem linhas repetidas com `model name`. O último pipe conecta `grep 'model name'` ao `uniq`, que é responsável por pular qualquer linha idêntica à anterior.

Os pipes podem ser combinados com redirecionamentos na mesma linha de comando. O exemplo anterior pode ser reescrito em uma forma mais simples:

```
$ grep 'model name' </proc/cpuinfo | uniq
model name      : Intel(R) Xeon(R) CPU           X5355  @ 2.66GHz
```

O redirecionamento de entrada para `grep` não é estritamente necessário, pois `grep` aceita um caminho de arquivo como argumento, mas o exemplo demonstra como construir comandos combinados.

Pipes e redirecionamentos são exclusivos, ou seja, uma origem pode ser mapeada para apenas um destino. Ainda assim, é possível redirecionar uma saída para um arquivo e ainda vê-la na tela com o programa `tee`. Para isso, o primeiro programa envia sua saída para o stdin de `tee` e um nome de arquivo é fornecido a este último para armazenar os dados:

```
$ grep 'model name' </proc/cpuinfo | uniq | tee cpu_model.txt
model name      : Intel(R) Xeon(R) CPU           X5355  @ 2.66GHz
$ cat cpu_model.txt
model name      : Intel(R) Xeon(R) CPU           X5355  @ 2.66GHz
```

A saída do último programa na cadeia, gerada por `uniq`, é exibida e armazenada no arquivo `cpu_model.txt`. Para não sobrescrever o conteúdo do arquivo fornecido, e sim anexar dados a ele, a opção `-a` deve ser fornecida para `tee`.

Apenas a saída padrão de um processo é capturada por um pipe. Digamos que você precise passar por um longo processo de compilação na tela e, ao mesmo tempo, salvar tanto a saída padrão quanto o erro padrão em um arquivo para inspeção posterior. Supondo que seu diretório atual não tenha um *Makefile*, o seguinte comando gerará um erro:

```
$ make | tee log.txt
make: *** No targets specified and no makefile found.  Stop.
```

Embora exibida na tela, a mensagem de erro gerada por `make` não foi capturada por `tee` e o arquivo *log.txt* foi criado vazio. É preciso fazer um redirecionamento antes que um pipe possa capturar o stderr:

```
$ make 2>&1 | tee log.txt
make: *** No targets specified and no makefile found.  Stop.
$ cat log.txt
make: *** No targets specified and no makefile found.  Stop.
```

Neste exemplo, o stderr de `make` foi redirecionado para o stdout, de forma que o `tee` foi capaz de capturá-lo com um pipe, exibi-lo na tela e salvá-lo no arquivo `log.txt`. Em casos como esse, pode ser útil salvar as mensagens de erro para inspeção posterior.

Substituição de comando

Outro método para capturar a saída de um comando é a *substituição de comando*. Ao colocar um comando entre crases, o Bash o substitui por sua saída padrão. O exemplo a seguir mostra como usar o stdout de um programa como argumento para outro programa:

```
$ mkdir `date +%Y-%m-%d`
$ ls
2019-09-05
```

A saída do programa `date`, a data atual formatada como *ano-mês-dia*, foi usada como um argumento para criar um diretório com o `mkdir`. Um resultado idêntico é obtido usando `$()` em vez de crases:

```
$ rmdir 2019-09-05
```

```
$ mkdir $(date +%Y-%m-%d)
$ ls
2019-09-05
```

O mesmo método pode ser usado para armazenar a saída de um comando como uma variável:

```
$ OS=`uname -o`
$ echo $OS
GNU/Linux
```

O comando `uname -o` retorna o nome genérico do sistema operacional atual, que foi armazenado na variável de sessão `OS`. Atribuir a saída de um comando a uma variável é muito útil em scripts, possibilitando armazenar e avaliar os dados de várias maneiras distintas.

Dependendo da saída gerada pelo comando substituído, a substituição do comando interno do shell pode não ser apropriada. Um método mais sofisticado para usar a saída de um programa como argumento de outro programa emprega um intermediário chamado `xargs`. O programa `xargs` usa o conteúdo que recebe via stdin para executar um determinado comando com o conteúdo como argumento. O exemplo a seguir mostra o `xargs` executando o programa `identify` com argumentos fornecidos pelo programa `find`:

```
$ find /usr/share/icons -name 'debian*' | xargs identify -format "%f: %wx%h\n"
debian-swirl.svg: 48x48
debian-swirl.png: 22x22
debian-swirl.png: 32x32
debian-swirl.png: 256x256
debian-swirl.png: 48x48
debian-swirl.png: 16x16
debian-swirl.png: 24x24
debian-swirl.svg: 48x48
```

O programa `identify` é parte do *ImageMagick*, um conjunto de ferramentas de linha de comando para inspecionar, converter e editar a maioria dos tipos de arquivo de imagem. No exemplo, o `xargs` pegou todos os caminhos listados por `find` e os colocou como argumentos para `identify`, que então exibe as informações para cada arquivo formatado conforme exigido pela opção `-format`. Os arquivos encontrados pelo `find` no exemplo são imagens contendo o logotipo da distribuição em um sistema de arquivos Debian. `-format` é um parâmetro para `identify`, não para `xargs`.

A opção `-n 1` exige que o `xargs` execute o comando fornecido com apenas um argumento por vez. No caso do exemplo, em vez de passar todos os caminhos encontrados por `find` como uma lista de

argumentos para `identify`, o uso de `xargs -n 1` executaria o comando `identify` para cada caminho separadamente. Usar `-n 2` executaria o `identify` com dois caminhos como argumentos, `-n 3` com três caminhos como argumentos e assim por diante. Da mesma forma, quando o `xargs` processa conteúdos com várias linhas — como é o caso com a entrada fornecida por `find` — a opção `-L` pode ser usada para limitar quantas linhas serão usadas como argumentos por execução do comando.

NOTE

Pode ser desnecessário usar o `xargs` com a opção `-n 1` ou `-L 1` para processar a saída gerada pelo `find`. O comando `find` tem a opção `-exec` para executar um comando determinado para cada item do resultado da busca.

Se os caminhos contiverem caracteres de espaço, é importante executar o `find` com a opção `-print0`. Esta opção instrui o `find` a usar um caractere nulo entre cada entrada para que a lista possa ser analisada corretamente por `xargs` (a saída foi suprimida):

```
$ find . -name '*avi' -print0 -o -name '*mp4' -print0 -o -name '*mkv' -print0 |
xargs -0 du | sort -n
```

A opção `-0` diz ao `xargs` que o caractere nulo deve ser usado como separador. Dessa forma, os caminhos de arquivo fornecidos pelo `find` são analisados corretamente, mesmo se contiverem espaços em branco ou outros caracteres especiais. O exemplo anterior mostra como usar o comando `du` para descobrir o uso de espaço em disco por cada arquivo encontrado e em seguida classificar os resultados por tamanho. A saída foi suprimida para fins de concisão. Observe que para cada critério de pesquisa é necessário incluir a opção `-print0` para `find`.

Por padrão, o `xargs` coloca por último os argumentos do comando executado. Para mudar esse comportamento, usamos a opção `-I`:

```
$ find . -mindepth 2 -name '*avi' -print0 -o -name '*mp4' -print0 -o -name '*mkv'
-print0 | xargs -0 -I PATH mv PATH ./
```

No último exemplo, todo arquivo encontrado por `find` é movido para o diretório atual. Como o(s) caminho(s) de origem devem ser informados para o `mv` antes do caminho de destino, um termo de substituição é dado à opção `-I` do `xargs`, que é então apropriadamente colocado junto a `mv`. Ao usar o caractere nulo como separador, não é necessário colocar o termo de substituição entre aspas.

Exercícios Guiados

1. É conveniente salvar a data de execução das ações realizadas por scripts automáticos. O comando `date +%Y-%m-%d` mostra a data atual no formato *ano-mês-dia*. Como a saída desse comando pode ser armazenada em uma variável do shell chamada `TODAY` usando a substituição de comando?

2. Usando o comando `echo`, como o conteúdo da variável `TODAY` pode ser enviado para a saída padrão do comando `sed s/-/./g`?

3. Como a saída do comando `date +%Y-%m-%d` pode ser usada como uma Here string para o comando `sed s/-/./g`?

4. O comando `convert image.jpeg -resize 25% small/image.jpeg` cria uma versão menor de `image.jpeg` e coloca a imagem resultante em um arquivo com o mesmo nome dentro do subdiretório `small`. Usando o `xargs`, como é possível executar o mesmo comando para todas as imagens listadas no arquivo `filelist.txt`?

Exercícios Exploratórios

1. Uma rotina de backup simples cria periodicamente uma imagem da partição `/dev/sda1` com `dd < /dev/sda1 > sda1.img`. Para realizar futuras verificações de integridade de dados, a rotina também gera um hash SHA1 do arquivo com `sha1sum < sda1.img > sda1.sha1`. Adicionando pipes e o comando `tee`, como esses dois comandos poderiam ser combinados em um só?

2. O comando `tar` é usado para empacotar muitos arquivos em um só, preservando a estrutura de diretórios. A opção `-T` permite especificar um arquivo contendo os caminhos a arquivar. Por exemplo, `find /etc -type f | tar -cJ -f /srv/backup/etc.tar.xz -T -` cria o arquivo tar compactado `etc.tar.xz` a partir da lista fornecida pelo comando `find` (a opção `-T -` indica a entrada padrão como a lista de caminhos). Para evitar possíveis erros de análise devido a caminhos que contêm espaços, quais opções deveriam estar presentes para os comandos `find` e `tar`?

3. Em vez de abrir uma nova sessão remota do shell, o comando `ssh` pode simplesmente executar um comando indicado como argumento: `ssh user@storage "remote command"`. Dado que `ssh` também permite redirecionar a saída padrão de um programa local para a entrada padrão do programa remoto, como o comando `cat` canalizaria um arquivo local chamado `etc.tar.gz` para `/srv/backup/etc.tar.gz` em `user@storage` através de `ssh`?

Resumo

Esta lição cobre as técnicas tradicionais de comunicação entre processos empregadas pelo Linux. A *canalização de comandos* cria um canal de comunicação unilateral entre dois processos e a *substituição de comandos* permite armazenar a saída de um processo em uma variável shell. A lição passa pelas seguintes etapas:

- Como *pipes* podem ser usados para transmitir a saída de um processo para a entrada de outro processo.
- A finalidade dos comandos `tee` e `xargs`.
- Como capturar a saída de um processo com a *substituição de comando*, armazenando-a em uma variável ou usando-a diretamente como parâmetro para outro comando.

Os comandos e procedimentos abordados foram:

- Canalização de comandos com `|`.
- Substituição de comandos com crases e `$()`.
- Os comandos `tee`, `xargs` e `find`.

Respostas aos Exercícios Guiados

1. É conveniente salvar a data de execução das ações realizadas por scripts automáticos. O comando `date +%Y-%m-%d` mostra a data atual no formato *ano-mês-dia*. Como a saída desse comando pode ser armazenada em uma variável do shell chamada `TODAY` usando a substituição de comando?

```
$ TODAY=`date +%Y-%m-%d`
```

ou

```
$ TODAY=$(date +%Y-%m-%d)
```

2. Usando o comando `echo`, como o conteúdo da variável `TODAY` pode ser enviado para a saída padrão do comando `sed s/-/./g`?

```
$ echo $TODAY | sed s/-/./g
```

3. Como a saída do comando `date +%Y-%m-%d` pode ser usada como uma Here string para o comando `sed s/-/./g`?

```
$ sed s/-/./g <<< `date +%Y-%m-%d`
```

ou

```
$ sed s/-/./g <<< $(date +%Y-%m-%d)
```

4. O comando `convert image.jpeg -resize 25% small/image.jpeg` cria uma versão menor de `image.jpeg` e coloca a imagem resultante em um arquivo com o mesmo nome dentro do subdiretório `small`. Usando o `xargs`, como é possível executar o mesmo comando para todas as imagens listadas no arquivo `filelist.txt`?

```
$ xargs -I IMG convert IMG -resize 25% small/IMG < filelist.txt
```

ou

```
$ cat filelist.txt | xargs -I IMG convert IMG -resize 25% small/IMG
```

Respostas aos Exercícios Exploratórios

1. Uma rotina de backup simples cria periodicamente uma imagem da partição `/dev/sda1` com `dd < /dev/sda1 > sda1.img`. Para realizar futuras verificações de integridade de dados, a rotina também gera um hash SHA1 do arquivo com `sha1sum < sda1.img > sda1.sha1`. Adicionando pipes e o comando `tee`, como esses dois comandos poderiam ser combinados em um só?

```
# dd < /dev/sda1 | tee sda1.img | sha1sum > sda1.sha1
```

2. O comando `tar` é usado para empacotar muitos arquivos em um só, preservando a estrutura de diretórios. A opção `-T` permite especificar um arquivo contendo os caminhos a arquivar. Por exemplo, `find /etc -type f | tar -cJ -f /srv/backup/etc.tar.xz -T -` cria o arquivo tar compactado `etc.tar.xz` a partir da lista fornecida pelo comando `find` (a opção `-T -` indica a entrada padrão como a lista de caminhos). Para evitar possíveis erros de análise devido a caminhos que contêm espaços, quais opções deveriam estar presentes para os comandos `find` e `tar`?

   Options `-print0` and `--null`:

```
$ find /etc -type f -print0 | tar -cJ -f /srv/backup/etc.tar.xz --null -T -
```

3. Em vez de abrir uma nova sessão remota do shell, o comando `ssh` pode simplesmente executar um comando indicado como argumento: `ssh user@storage "remote command"`. Dado que `ssh` também permite redirecionar a saída padrão de um programa local para a entrada padrão do programa remoto, como o comando `cat` canalizaria um arquivo local chamado `etc.tar.gz` para `/srv/backup/etc.tar.gz` em `user@storage` através de `ssh`?

```
$ cat etc.tar.gz | ssh user@storage "cat > /srv/backup/etc.tar.gz"
```

   or

```
$ ssh user@storage "cat > /srv/backup/etc.tar.gz" < etc.tar.gz
```