Awk

Содержание

Последнее редактирование: Пт Ноя 27 09:56:48 2020

Часть учебных пособий по Unix, а затем есть мой блог

Вы можете угостить меня кофе, пожалуйста

Я был бы признателен, если бы вы иногда угощали меня кофе. Спасибо.

Нажмите на тему в этой таблице, чтобы перейти туда. Нажмите на название темы, чтобы вернуться к оглавлению.

Зачем изучать AWK?
Базовая структура
Выполнение скрипта AWK
Какую оболочку использовать с AWK?
Динамические переменные
Основной синтаксис AWK
Арифметические выражения
  Унарные арифметические операторы
  Операторы автоматического увеличения и автоматического уменьшения
  Операторы присваивания
  Условные выражения
  Регулярные выражения
  И / Или / Не
Краткое описание команд AWK
Встроенные переменные AWK
  FS - переменная разделителя полей ввода
  OFS - переменная разделителя выходных полей
  NF - переменная количества полей
  NR - переменная количества записей
  RS - переменная-разделитель записей
  ORS - переменная разделителя выходных записей
  FILENAME - текущая переменная Filename
Ассоциативные массивы
  Многомерные массивы
  Пример использования ассоциативных массивов AWK
    Вывод сценария
Идеальный вывод PRINTF
  PRINTF - форматирование вывода
  Escape-последовательности
  Спецификаторы формата
  Ширина - указание минимального размера поля
  Выравнивание по левому краю
  Значение точности поля
  Явный вывод файла
Управление потоком с помощью next и exit

Числовые функции AWK
Тригонометрические функции
Экспоненты, логарифмы и квадратные корни
Усечение целых чисел
Случайные числа
Сценарий лотереи
Строковые функции
Функция длины
Индексная функция
Функция Substr
Функция разделения
Функция GAWK's Tolower и Toupper
Строковые функции NAWK
Функция сопоставления
Системная функция
Функция Getline
Функция системного времени
Функция Strftime
Определяемые пользователем функции
Шаблоны AWK
Форматирование программ AWK
Переменные среды
ARGC - число или аргументы (NAWK / GAWK)
ARGV - массив аргументов (NAWK / GAWK)
ARGIND - индекс аргументов (только для GAWK)
FNR (NAWK / GAWK)
OFMT (NAWK / GAWK)
ПЕРВЫЙ ЗАПУСК, ДЛИНА и соответствие (NAWK / GAWK)
SUBSEP - разделитель многомерных массивов (NAWK / GAWK)
ENVIRON - переменные среды (только для GAWK)
ИГНОРИРОВАНИЕ (только для GAWK)
CONVFMT - формат преобразования (только для GAWK)
ERRNO - системные ошибки (только для GAWK)
FIELDWIDTHS - поля фиксированной ширины (только для GAWK)
AWK, NAWK, GAWK или PERL

Введение в AWK

Оригинальная версия написана в 1994 году и опубликована в Sun Observer

Awk - чрезвычайно универсальный язык программирования для работы с файлами. Мы научим вас достаточно, чтобы понять примеры на этой странице, плюс немного.

Приведенные ниже примеры содержат расширения исполняемого скрипта как часть имени файла. Как только вы загрузите его и сделаете его исполняемым, вы можете переименовать его как угодно.

Зачем изучать AWK?

В прошлом я рассматривал *grep* и *sed*. В этом разделе обсуждается AWK, еще один краеугольный камень программирования оболочки UNIX. Существует три варианта AWK:

AWK - (очень старый) оригинал от AT & T
NAWK - более новая, улучшенная версия от AT & T
GAWK - версия фонда свободного программного обеспечения

Изначально я не планировал обсуждать NAWK, но несколько поставщиков UNIX заменили AWK на NAWK, и между ними есть несколько несовместимостей. Было бы жестоко с моей стороны не предупредить вас о различиях. Поэтому я выделю их, когда дойду до них. Важно знать, что все функции AWK находятся в NAWK и GAWK. Большинство, если не все, функций NAWK находятся в GAWK. NAWK поставляется как часть Solaris. GAWK этого не делает. Тем не менее, многие сайты в Интернете имеют источники в свободном доступе. Если вы используете Linux, у вас есть GAWK. Но целом, предположим, что я говорю о классическом AWK, если не указано иное.

А теперь поговорим о MAWK, TAWK и JAWK.

Почему AWK так важен? Это отличный инструмент для создания фильтров и отчетов. Многие утилиты UNIX генерируют строки и столбцы информации. AWK - отличный инструмент для обработки этих строк и столбцов, и использовать AWK проще, чем большинство обычных языков программирования. Его можно рассматривать как интерпретатор псевдо-C, поскольку он понимает те же арифметические операторы, что и C. AWK также имеет функции обработки строк, поэтому он может искать определенные строки и изменять выходные данные. AWK также имеет ассоциативные массивы, которые невероятно полезны и являются особенностью, которой не хватает большинству вычислительных языков. Ассоциативные массивы могут превратить сложную задачу в тривиальное упражнение.

Я постараюсь охватить основные части AWK и упомянуть расширения / варианты. "Новый AWK", ил "nawk", поставляется в системе Sun, и вы можете обнаружить, что он превосходит старый AWK во многих отношениях. В частности, он имеет лучшую диагностику и не будет выводить печально известное сообщение "выход из строя рядом с линией ...", к которому склонен оригинальный AWK. Вместо этого "nawk" выводит строку, которую он не понял, и выделяет плохие части стрелками. GAWK тоже это делает, и это действительно очень помогает. Если вам понадобится функция, которую очень сложно или невозможно выполнить в AWK, я предлагаю вам либо использовать NAWK, либо GAWK, либо преобразовать ваш AWK-скрипт в PERL с помощью программы преобразования "a2p", которая поставляется с PERL. PERL - замечательный язык, и я использую его постоянно, но я не планирую освещать PERL в этих уроках. Прояснив свое намерение, я могу продолжать с чистой совестью.

Многие утилиты UNIX имеют странные имена. AWK является одной из таких утилит. Это не сокращение от *awk* ward. На самом деле, это элегантный и простой язык. Название "AWK" происходит от инициалов трех разработчиков языка: А. Ахо, Б. В. Кернигана и П. Вайнбергера.

Базовая структура

Основная организация программы AWK соответствует форме:

шаблон { действие}

Шаблон указывает, когда выполняется действие. Как и большинство утилит UNIX, AWK ориентирована на строки. То есть шаблон определяет тест, который выполняется с каждой строкой, считываемой в качестве входных данных. Если условие истинно, то действие выполняется. Шаблон по умолчанию - это то, что соответствует каждой строке. Это пустой или нулевой шаблон. Два други важных шаблона определяются ключевыми словами "BEGIN" и "END". Как и следовало ожидать, эти два слова определяют действия, которые необходимо предпринять перед чтением любых строк и после чтения последней строки. Программа AWK ниже:

```
НАЧАТЬ { напечатать "НАЧАТЬ"}
 { печать }
ЗАВЕРШЕНИЕ { печать "СТОП" }
```

добавляет одну строку до и одну строку после входного файла. Это не очень полезно, но с помощью простого изменения мы можем превратить это в типичную программу AWK:

```
НАЧАТЬ { напечатать "File \ tOwner"}
{ напечатать $ 8, "\ t", $ 3}
КОНЕЦ { напечатать " - ГОТОВО -" }
```

Я улучшу сценарий в следующих разделах, но мы будем называть его "FileOwner". Но давайте пока не будем помещать это в сценарий или файл. Я немного расскажу об этой части. Держитесь и следуйте за мной, чтобы вы почувствовали вкус AWK.

Символы "\ t" указывают на символ табуляции, поэтому выходные данные выстраиваются в четные границы. "$ 8" и "$ 3" имеют значение, аналогичное сценарию оболочки. Вместо восьмого и третьего аргумента они означают восьмое и третье поля строки ввода. Вы можете представить поле как столбец, и указанное вами действие будет действовать на каждую прочитанную строку или строку.

Есть два различия между AWK и оболочкой, обрабатывающей символы в двойных кавычках. AWK понимает, что специальные символы следуют за символом "\", таким как "t". Оболочки Bourne и C UNIX этого не делают. Кроме того, в отличие от оболочки (и PERL) AWK не оценивает переменные внутри строк. Чтобы объяснить, вторая строка не может быть написана так:

```
{вывести "$ 8 \ t $ 3" }
```

В этом примере будет напечатано "$ 8 $ 3". Внутри кавычек знак доллара не является специальным символом. Снаружи это соответствует полю. Что я подразумеваю под третьим и восьмым полями? Рассмотрим команду Solaris "/ usr / bin / ls -l", которая содержит восемь столбцов информации. Версия System V (аналогичная версии Linux), "/usr/5bin/ls -l" имеет 9 столбцов. Третий столбец - это владелец, а восьмой (или девятый) столбец в имени файла. Эта программа AWK может использоваться для обработки выходных данных команды "ls -l", распечатывая имя файла, а затем владельца для каждого файла. Я покажу вам, как.

Обновление: в системе Linux измените "$ 8" на "$ 9".

Еще один момент об использовании знака доллара. В скриптовых языках, таких как Perl и различных оболочках, знак доллара означает, что слово, следующее за именем переменной. Awk отличается. Знак доллара означает, что мы ссылаемся на поле или столбец в текущей строке. При переключении между Perl и AWK вы должны помнить, что "$" имеет другое значение. Итак, следующий фрагмент кода печатает два "поля" для стандартного вывода. Первое напечатанное поле - это число "5", второе - пятое поле (или столбец) в строке ввода.

```
НАЧАТЬ { x=5 }
{ вывести x, $x}
```

Выполнение скрипта AWK

Итак, давайте начнем писать наш первый сценарий AWK. Есть несколько способов сделать это.

Предполагая, что первый сценарий называется "FileOwner", вызов будет

```
ls -l | FileOwner
```

Это могло бы привести к следующему, если бы в текущем каталоге было только два файла:

```
Владелец файла


a.файл Барнетта
другой.файл Барнетт
- ГОТОВО -
```

С этим скриптом есть две проблемы. Обе проблемы легко исправить, но я воздержусь от этого, пока не рассмотрю основы.

Сам сценарий может быть написан многими способами. Я показал как оболочку C (csh / tcsh), так и сценарий оболочки Bourne / Bash / POSIX. Версия оболочки C будет выглядеть следующим образом

```
#!/bin/csh -f
# Пользователи Linux должны изменить awk с 8 на 9 долларов
'\
```

```
НАЧАТЬ { распечатать "File\tOwner" } \
 { вывести $ 8, "\ t", $ 3} \
КОНЕЦ { печать " - ГОТОВО -" } \
'
```

И, конечно, после создания этого скрипта вам нужно сделать этот скрипт исполняемым, набрав

```
chmod +x awk_example.1.csh
```

Нажмите здесь, чтобы получить файл: awk_example1.csh

Как вы можете видеть в приведенном выше сценарии, каждая строка сценария AWK должна иметь обратную косую черту, если это не последняя строка сценария. Это необходимо, поскольку оболочка C по умолчанию не допускает, чтобы строки были длиннее строки. У меня есть длинный список жалоб на использование оболочки C. Смотрите Десять основных причин не использовать оболочку C
Оболочка Bourne (как и большинство оболочек) позволяет заключенным в кавычки строкам занимать несколько строк:

```
#!/bin/sh
# Пользователи Linux должны изменить $ 8 на $ 9
awk '
BEGIN { print "File\tOwner" }
{ print $ 8, "\ t", $ 3}
END { print " - DONE -" }
'
```

И снова, как только он создан, он должен быть выполнен:

```
chmod +x awk_example1.sh
```

Нажмите здесь, чтобы получить файл: awk_example1.sh

Кстати, я привожу примеры сценариев в учебном пособии и использую расширение имени файла для указания типа сценария. Вы, конечно, можете "установить" скрипт в свой домашний каталог "bin набрав

```
cp awk_example1.sh $HOME/bin/неудобный
пример1 chmod +x $HOME/bin/неудобный пример1
```

Третий тип AWK-скриптов - это "родной" AWK-скрипт, в котором вы не используете оболочку. Вы можете записать команды в файл и выполнить

```
имя файла awk -f
```

Поскольку AWK также является интерпретатором, как и оболочка, вы можете сэкономить шаг и сделать файл исполняемым, добавив одну строку в начало файла:

```
#!/bin/awk -f
BEGIN { печать "File \tOwner" }
{ печать $ 8, "\ t", $ 3}
END { печать " - ГОТОВО -" }
```

Затем выполните "chmod + x" и используйте этот файл как новую команду UNIX.
Нажмите здесь, чтобы получить файл: awk_example1.awk

Обратите внимание на опцию "-f", следующую за "#!/bin / awk " выше, которая также используется в третьем формате, где вы используете AWK для непосредственного выполнения файла, то есть "awk -f filename". Параметр "-f" указывает файл AWK, содержащий инструкции. Как вы можете видеть, AWK рассматривает строки, начинающиеся с "#", как комментарий, точно так же, как оболочка. Чтобы быть точным, все, что от "#" до конца строки, является комментарием (если только оно не находится внутри строки AWK. Тем не менее, я всегда комментирую свои сценарии AWK с помощью "#" в начале строки по причинам, которые я рассмотрю позже.

Какой формат вы должны использовать? Я предпочитаю последний формат, когда это возможно. Это короче и проще. Это также упрощает отладку проблем. Если вам нужно использовать оболочку вы хотите избежать использования слишком большого количества файлов, вы можете объединить их, как мы делали в первом и втором примере.

Какую оболочку использовать с AWK?

Формат оригинального AWK не является произвольной формой. Вы не можете ставить новые разрывы строк где попало. Они должны находиться в определенных местах. Чтобы быть точным, в оригинальном AWK вы можете вставить символ новой строки после фигурных скобок и в конце команды, но не в другом месте. Если вы хотели разбить длинную строку на две строки в любом другом месте, вам нужно было использовать обратную косую черту:

```
#!/bin/awk -f
BEGIN { печать "File \tOwner" }
{ печать $ 8, "\ t", \
$ 3}
END { печать " - ГОТОВО -" }
```

Нажмите здесь, чтобы получить файл: awk_example2.awk

Версия Bourne shell будет

```
#!/bin/sh
awk '
BEGIN { печать "File\tOwner" }
{ печать $ 8, "\ t", \
$ 3}
КОНЕЦ { печать "готово"}
'
```

Нажмите здесь, чтобы получить файл: awk_example2.sh

в то время как оболочка C будет

```
#!/bin/csh -f
awk '
BEGIN { печать "File\tOwner" }\
{ печать $ 8, "\ t", \\
$ 3} \
END { печать "готово"} \
'
```

Нажмите здесь, чтобы получить файл: awk_example2.csh

Как вы можете видеть, это демонстрирует, насколько неудобна оболочка C при вложении скрипта AWK. Для каждой строки нужны не только обратные косые черты, для некоторых строк требуется две, тогда используется старый оригинальный AWK. Новые AWK более гибкие, в них можно добавлять новые строки. Многие люди, как и я, предупредят вас о оболочке C. Некоторые проблемы неуловимы, и вы можете никогда их не увидеть. Попробуйте включить сценарий AWK или *sed* в сценарий оболочки C, и обратные косые черты сведут вас с ума. Это то, что убедило меня изучить оболочку Bourne много лет назад, когда я только начинал (до того, как были доступны оболочки Korn или Bash). Даже если вы настаиваете на использовании оболочки C, вы должны, по крайней мере, достаточно изучить оболочку Borne / POSIX, чтобы задавать переменные, что по какому-то странному совпадению является предметом следующего раздела.

Динамические переменные

Поскольку вы можете сделать скрипт исполняемым в формате AWK, указав "#!/bin/awk -f" в первой строке, включение скрипта AWK в сценарий оболочки не требуется, если вы не хотите либо устранить необходимость в дополнительном файле, либо если вы хотите передать переменную ввнутренности скрипта AWK. Поскольку это распространенная проблема, сейчас самое подходящее

время объяснить технику. Я сделаю это, показав простую программу AWK, которая будет печатать только один столбец. **ПРИМЕЧАНИЕ: в первой версии будет ошибка.**Номер столбца будет указан первым аргументом. Первая версия программы, которую мы будем называть "Колонка", выглядит так:

```
#!/bin/sh
#ПРИМЕЧАНИЕ - этот скрипт не работает!
column="$ 1"
awk '{print $column}'
```

Нажмите здесь, чтобы получить файл (но имейте в виду, что это не работает): Column1.sh

Рекомендуемое использование:

```
ls -l | Колонка 3
```

Это приведет к печати третьего столбца из команды */s*, который будет владельцем файла. Вы может превратить это в утилиту, которая подсчитывает, сколько файлов принадлежит каждому пользователю, добавив

```
ls -l | Столбец 3 | uniq -c | sort -nr
```

Только одна проблема: скрипт не работает.Значение переменной "column" не отображается AWK Измените "awk" на "echo", чтобы проверить. Вам нужно отключить цитирование, когда отображается переменная. Это можно сделать, завершив цитирование и перезапустив его после переменной:

```
#!/bin/sh
column="$ 1"
awk '{print $"$column"}'
```

Нажмите здесь, чтобы получить файл: Column2.sh

Это очень важная концепция, которая ставит опытных программистов в тупик. Во многих компьютерных языках строка имеет начальную и конечную кавычки, а также промежуточное содержимое. Если вы хотите включить специальный символ внутри цитаты, вы должны запретить символу иметь типичное значение. В языке C это можно сделать, поставив обратную косую черту перед символом. В других языках для этого существует специальная комбинация символов. В оболочке C и Bourne цитата - это просто переключатель. Он включает или выключает режим интерпретации. На самом деле нет такого понятия, как "начало строки" и "конец строки". Кавычки переключают переключатель внутри интерпретатора. Символ кавычки не передается приложению. Вот почему выше приведены две пары цитат. Обратите внимание, что есть два знака доллара. Первый цитируется и просматривается AWK. Второй не заключен в кавычки, поэтому оболочка оценивает переменную и заменяет "$column" значением. Если вы не понимаете, либо измените "awk" на "echo", либо измените первую строку на "#! / bin / sh -x".

Однако необходимы некоторые улучшения. Оболочка Bourne имеет механизм для предоставления значения переменной, если значение не задано или установлено, а значение представляет собой пустую строку. Это делается с помощью формата:

```
${переменная: -значение по умолчанию}
```

Это показано ниже, где столбцом по умолчанию будет один:

```
#!/bin/sh
column="${1: -1}"
awk '{print $"$column"}'
```

Нажмите здесь, чтобы получить файл: Column3.sh

Мы можем сохранить строку, объединив эти два шага:

```
#!/bin/sh
awk '{печать $"${1:-1}"}'
```

Нажмите здесь, чтобы получить файл: Column4.sh

Это трудно читать, но это компактно. Есть еще один метод, который можно использовать. Если вы выполните команду AWK и включите в командную строку информацию в следующей форме:

переменная = значение

эта переменная будет установлена при запуске скрипта AWK. Примером такого использования может быть:

```
#!/bin/sh
awk '{print $c}' c="${1:-1}"
```

Нажмите здесь, чтобы получить файл: Column5.sh

У этого последнего варианта нет проблем с цитированием предыдущего примера. Однако вы должны освоить предыдущий пример, потому что вы можете использовать его с любым сценарием или командой. Второй метод является специальным для AWK. У современных AWK есть и другие варианты. См. FAQ по comp.unix.shell.

Основной синтаксис AWK

Ранее я обсуждал способы запуска скрипта AWK. В этом разделе будут рассмотрены различные грамматические элементы AWK.

Арифметические выражения

Существует несколько арифметических операторов, похожих на C. Это бинарные операторы, которые работают с двумя переменными:

Бинарные операторы AWK таблица 1		
Оператор	**Тип**	**Значение**
+	Арифметика	Дополнение
-	Арифметика	Вычитание
*	Арифметика	Умножение
/	Арифметика	Подразделение
%	Арифметика	Modulo
<пробел>	Строка	Объединение

Используя переменные со значением "7" и "3", AWK возвращает следующие результаты для каждого оператора при использовании команды печати:

Выражение	**Результат**
7+3	10
7-3	4
7*3	21
7/3	2.33333
7%3	1
7 3	73

Необходимо сделать несколько замечаний. Оператор модуля находит остаток после деления на целое число. Команда *print* выводит число с плавающей запятой при разделении, но целое число дл остальных. Оператор конкатенации строк сбивает с толку, поскольку он даже не виден. Поместите пробел между двумя переменными, и строки будут объединены вместе. Это также показывает, что числа автоматически преобразуются в строки, когда это необходимо. В отличие от C, AWK не имеет "типов" переменных. Существует только один тип, и это может быть строка или число. Правила преобразования просты. Число можно легко преобразовать в строку. Когда строка преобразуется в число, AWK сделает это. Строка "123" будет преобразована в число 123. Однако строка "123X" буде

преобразована в число 0. (NAWK будет вести себя иначе и преобразует строку в целое число 123, которое находится в начале строки).

Унарные арифметические операторы

Операторы "+" и "-" могут использоваться перед переменными и числами. Если X равно 4, то утверждение:

```
печать -x;
```

будет напечатано "-4".

Операторы автоматического увеличения и автоматического уменьшения

AWK также поддерживает операторы "++" и "--" C. Оба увеличивают или уменьшают переменные на единицу. Оператор может использоваться только с одной переменной и может быть до или после переменной. Префиксная форма изменяет значение, а затем использует результат, в то время как постфиксная форма получает результаты переменной, а затем изменяет переменную. В качестве примера, если X имеет значение 3, то оператор AWK

```
выведите x++, " ", ++x;
```

напечатал бы цифры 3 и 5. Эти операторы также являются операторами присваивания и могут использоваться сами по себе в строке:

```
x++;
--y;
```

Операторы присваивания

Переменным можно присваивать новые значения с помощью операторов присваивания. Вы знаете о "++" и "--". Другой оператор присваивания просто:

```
переменная = arithmetic_expression
```

Некоторые операторы имеют приоритет над другими; скобки можно использовать для управления группировкой. Заявление

```
x =1+2*3 4;
```

это то же самое, что

```
x = (1 + (2 * 3)) "4";
```

Оба выводят "74".

Для удобства чтения можно добавить пробелы. AWK, как и C, имеет специальные операторы присваивания, которые объединяют вычисления с присваиванием. Вместо того, чтобы сказать

```
x = x +2;
```

вы можете более кратко сказать:

```
x+=2;
```

Ниже приведен полный список:

AWK Таблица 2
Операторы присваивания

Оператор	Значение
+=	Добавить результат в переменную
-=	Вычесть результат из переменной
*=	Умножение переменной на результат
/=	Разделите переменную на результат
%=	Применить по модулю к переменной

Условные выражения

Второй тип выражения в AWK - это условное выражение. Это используется для определенных тестов, таких как *if* или *while* . Логические условия оцениваются как true или false. В AWK существует определенная разница между логическим условием и арифметическим выражением. Вы не можете преобразовать логическое условие в целое число или строку. Однако вы можете использовать арифметическое выражение в качестве условного выражения. Значение 0 равно false, в то время ка все остальное равно true. Неопределенные переменные имеют значение 0. В отличие от AWK, NAWK позволяет использовать логические значения как целые числа.

Арифметические значения также могут быть преобразованы в логические условия с помощью реляционных операторов:

Таблица AWK 3 Операторы отношений	
Оператор	**Значение**
==	Равно
!=	Не равно
>	Больше, чем
>=	Больше или равно
<	Меньше, чем
<=	Меньше или равно

Эти операторы такие же, как операторы C. Их можно использовать для сравнения чисел или строк. Что касается строк, буквы нижнего регистра больше, чем буквы верхнего регистра.

Регулярные выражения

Для сравнения строк с регулярными выражениями используются два оператора:

AWK Таблица 4 Операторы регулярных выражений	
Оператор	**Значение**
~	Матчи
!~	Не соответствует

Порядок в этом случае особый. Регулярное выражение должно быть заключено в косую черту и находится после оператора. AWK поддерживает расширенные регулярные выражения, поэтому ниже приведены примеры допустимых тестов:

```
слово !~ /START/
lawrence_welk ~ /(один | два | три)/
```

И / Или / Не

Есть два логических оператора, которые можно использовать с условными выражениями. То есть вы можете комбинировать два условных выражения с операторами "или" или "и": "&&" и "||". Существуе также унарный оператор not: "!".

Краткое описание команд AWK

В AWK всего несколько команд. Список и синтаксис приведены ниже:

```
оператор if (условный) [оператор else ]
while (условный) оператор
для ( выражение; условное; выражение) заявление
оператор for ( переменная в массиве)
прервать
продолжение
{ [ заявление] ...}
переменная = выражение
печать [список выражений] [ > выражение]
формат printf [ , список выражений] [ > выражение]
```

следующий
выход

На этом этапе вы можете использовать AWK как язык для простых вычислений; Если вы хотите что-то вычислить, а не читать какие-либо строки для ввода, вы можете использовать ключевое слово *BEGIN*, рассмотренное ранее, в сочетании с командой *exit*:

```
#!/bin/awk -f
НАЧАТЬ {

# Выведите квадраты от 1 до 10 первым способом

 i = 1;
 в то время как (i <= 10) {
 printf "Квадрат ", i, " есть", i * i;
 i = i +1;
 }

# сделайте это снова, используя более сжатый код

 для (i=1; i <= 10; i++) {
 printf "Квадрат ", i, " есть", i * i;
 }

# теперь завершите
выход;
}
```

Нажмите здесь, чтобы получить файл: awk_print_squares.awk
Следующее запрашивает число, а затем помещает его в квадрат:

```
#!/bin/awk -f
НАЧАТЬ {
 выведите "введите число";
}
{
 выведите "Квадрат", $ 1, "равно", $ 1 * $ 1;
 выведите "введите другое число".;
}
КОНЕЦ {
 вывести "Готово"
}
```

Нажмите здесь, чтобы получить файл: awk_ask_for_square.awk

Приведенный выше не является хорошим фильтром, потому что он каждый раз запрашивает ввод. Если вы передадите в него выходные данные другой программы, вы сгенерируете множество бессмысленных подсказок.

Вот фильтр, который вы должны найти полезным. Он подсчитывает строки, суммирует числа в первом столбце и вычисляет среднее значение. Вставьте в него "wc -c *", и он подсчитает файлы и сообщит вам среднее количество слов в файле, а также общее количество слов и количество файлов.

```
#!/bin/awk -f
НАЧАТЬ {
# Сколько строк
 строки = 0;
 итого = 0;
}
{
# этот код выполняется один раз для каждой строки
# увеличить количество файлов
 строки ++;
# увеличьте общий размер, который равен полю № 1
итого + = $ 1;
}
```

```
КОНЕЦ {
# end, теперь выведите общее
количество строк печати "строки прочитаны";
 выведите "итого равно ", итого;
если (строки> 0) {
 выведите "среднее значение", всего / строк;
 } еще {
 выведите "среднее значение равно 0".;
 }
}
```

Нажмите здесь, чтобы получить файл: average.awk

Вы можете передать выходные данные "ls -s" в этот фильтр, чтобы подсчитать количество файлов, общий размер и средний размер. С этим скриптом есть небольшая проблема, поскольку он включает в себя вывод "ls", который сообщает общее количество. Это приводит к уменьшению количества файлов на единицу. Изменение

```
строки ++;
```

Для

```
если ($ 1!= "всего") строк ++;
```

исправит эту проблему. Обратите внимание на код, который предотвращает деление на ноль. Это часто встречается в хорошо написанных сценариях. Я также инициализирую переменные равными нулю. В этом нет необходимости, но это хорошая привычка.

Встроенные переменные AWK

Я упомянул два вида переменных: позиционные и определяемые пользователем. Пользовательская переменная - это та, которую вы создаете. Позиционная переменная - это не специальная переменная, а функция, запускаемая знаком доллара. Поэтому

```
распечатать 1 доллар;
```

и

```
X=1;
вывести $ X;
```

сделайте то же самое: выведите первое поле в строке. Есть еще два очень полезных пункта о позиционных переменных. Переменная "$ 0" относится ко всей строке, которую AWK считывает. То есть, если у вас было восемь полей в строке,

```
распечатать 0 долларов;
```

похож на

```
печать $1, $2, $3, $4, $5, $6, $7, $8
```

Это изменит интервал между полями; в противном случае они будут вести себя одинаково. Вы можете изменять позиционные переменные. Следующие команды

```
$ 2 = "";
распечатать;
```

удаляет второе поле. Если у вас есть четыре поля, и вы хотите распечатать второе и четвертое поля, есть два способа. Это первый:

```
#!/bin/awk -f
{
 $1="";
 $3="";
 Печать;
}
```

и второй

```
#!/bin/awk -f
{
 вывести 2, 4 доллара США;
}
```

Они работают аналогично, но не одинаково. Количество пробелов между значениями различно. Для этого есть две причины. Фактическое количество полей не меняется. Установка позиционной переменной в пустую строку не приводит к удалению переменной. Он все еще там, но содержимое было удалено. Другая причина заключается в том, как AWK выводит всю строку. Существует разделитель полей, который указывает, какой символ поместить между полями при выводе. В первом примере выводится четыре поля, а во втором - два. Между каждым полем находится пробел. Это легче объяснить, если символы между полями можно изменить, чтобы сделать их более заметными Что ж, это возможно. AWK предоставляет специальные переменные именно для этой цели.

FS - переменная разделителя полей ввода

AWK можно использовать для анализа многих файлов системного администрирования. Однако многие из этих файлов не имеют пробелов в качестве разделителя. в качестве примера в файле паролей используются двоеточия. Вы можете легко изменить символ разделителя полей на двоеточие, используя параметр командной строки "-F". Следующая команда распечатает учетные записи, у которых нет паролей:

```
awk -F: '{if ($ 2 == "") print $ 1 ": пароля нет!"}'
```

Есть способ сделать это без опции командной строки. Переменная "FS" может быть установлена ка любая переменная и имеет ту же функцию, что и параметр командной строки "-F". Ниже приведен скрипт, который выполняет ту же функцию, что и приведенный выше.

```
#!/bin/awk -f
НАЧАТЬ {
 FS=":";
}
{
 если ( $ 2 == "" ) {
 выведите 1 доллар ": без пароля!";
 }
}
```

Нажмите здесь, чтобы получить файл: awk_nopasswd.awk

Вторая форма может быть использована для создания утилиты UNIX, которую я назову "chkpasswd" и выполняется следующим образом:

```
chkpasswd
```

Команда "chkpasswd -F:" не может быть использована, потому что AWK никогда не увидит этот аргумент. Все сценарии интерпретатора принимают один и только один аргумент, который находится сразу после строки "#!/bin/awk". В этом случае единственным аргументом является "-f". Еще одним отличием между параметром командной строки и внутренней переменной является возможность установки разделителя полей ввода более чем на один символ. Если вы укажете

```
FS=": ";
```

затем AWK разделит строку на поля везде, где он видит эти два символа, в точном порядке. Вы не можете сделать это в командной строке.

Существует третье преимущество внутренней переменной перед параметром командной строки: вы можете изменять символ разделителя полей столько раз, сколько захотите, во время чтения файла. Ну, не более одного раза для каждой строки. Вы даже можете изменить его в зависимости от прочитанной строки. Предположим, у вас есть следующий файл, который содержит числа от 1 до 7

трех разных форматах. В строках с 4 по 6 поля разделены двоеточием, а остальные разделены пробелами.

```
ОДИН 1 I
ДВА 2 II
# НАЧАТЬ
ТРИ: 3: III
ЧЕТЫРЕ: 4: IV
ПЯТЬ: 5: V
# ОСТАНОВИТЬ
ШЕСТЬ 6 VI
СЕМЬ 7 VII
```

Программа AWK может легко переключаться между этими форматами:

```
#!/bin/awk -f
{
 если ($ 1 == "#START") {
 FS=":";
 } еще, если ($ 1 == "#STOP") {
 FS =" ";
 } еще {
 #выведите римское число в столбце 3
выведите 3 доллара
 }
}
```

Нажмите здесь, чтобы получить файл: awk_example3.awk

Обратите внимание, что переменная-разделитель полей сохраняет свое значение до тех пор, пока оно не будет явно изменено. Вам не нужно сбрасывать его для каждой строки. Звучит просто, не так ли? Тем не менее, у меня есть к вам вопрос с подвохом. Что произойдет, если вы измените разделитель полей во время чтения строки? То есть, предположим, у вас была следующая строка

```
Один Два: Три: 4 Пять
```

и вы выполнили следующий сценарий:

```
#!/bin/awk -f
{
 распечатать 2 доллара
 FS=":"
 распечатать 2 доллара
}
```

Что будет напечатано? "Три" или "Два: три: 4?" Ну, сценарий напечатал бы "Два: три: 4" дважды. Однако, если вы удалили первую инструкцию print, она напечатает "Три" один раз! Сначала я подумал, что это очень странно, но после того, как я вырвал несколько волос, пнул колоду и накричал на muself и всех, кто имел какое-либо отношение к разработке UNIX, это интуитивно очевидно. Вам просто нужно думать как профессиональный программист, чтобы понять, что это интуитивно понятно. Я объясню и помешаю вам причинить себе физический вред.

Если вы измените разделитель полей **перед** чтением строки, это изменение **повлияет** на то, что вы читаете. Если вы измените его **после** прочтения строки, это **не** приведет к переопределению переменных. Вы бы не хотели, чтобы переменная изменялась у вас как побочный эффект другого действия. Язык программирования со скрытыми побочными эффектами сломан, и ему не следует доверять. AWK позволяет переопределить разделитель полей либо до, либо после прочтения строки и каждый раз делает все правильно. После того, как вы прочитаете переменную, переменная не изменится, если вы ее не измените. Браво!

Чтобы проиллюстрировать это далее, вот еще одна версия предыдущего кода, которая динамически изменяет разделитель полей. В этом случае AWK делает это, исследуя поле "$ 0", которое представляет собой всю строку. Когда строка содержит двоеточие, разделителем полей является двоеточие, в противном случае это пробел. Вот версия, которая работала со старыми версиями awk

```
#!/bin/awk -f
{
 если ( $ 0 ~ /:/) {
 FS=":";
 } еще {
 FS =" ";
 }
 #распечатать третье поле, в любом формате
распечатать $ 3
}
```

Нажмите здесь, чтобы получить файл: awk_example4.awk

Однако это поведение изменилось в более поздних версиях, поэтому приведенный выше сценарий больше не работает. Что происходит, так это то, что после изменения переменной FS вам необходимо повторно оценить поля, используя *$ 0 = $ 0*:

```
#!/bin/awk -f
{
 если ( $ 0 ~ /:/) {
 FS=":";
 $0=$0
 } еще {
 FS =" ";
 $0=$0
 }
 #распечатать третье поле, в любом формате
распечатать $ 3
}
```

Нажмите здесь, чтобы получить файл: awk_example4a.awk

Этот пример устраняет необходимость иметь специальные строки "#START" и "#STOP" во входных данных.

OFS - переменная разделителя выходных полей

Существует важное различие между

> вывести $ 2 $ 3

и

> распечатать 2, 3 доллара

В первом примере выводится одно поле, а во втором - два поля. В первом случае два позиционных параметра объединяются вместе и выводятся без пробела. Во втором случае AWK печатает два поля и помещает разделитель выходных полей между ними. Обычно это пробел, но вы можете изменить его, изменив переменную "OFS".

Если вы хотите скопировать файл паролей, но удалить зашифрованный пароль, вы можете использовать AWK:

```
#!/bin/awk -f
НАЧАТЬ {
 FS=":";
 OFS=":";
}
{
 $2="";
 Печать
}
```

Нажмите здесь, чтобы получить файл: delete_passwd.awk

Предоставьте этому скрипту файл паролей, и он удалит пароль, но оставит все остальное без изменений. Вы можете сделать разделитель полей вывода любым количеством символов. Вы не ограничены одним персонажем.

NF - переменная количества полей

Полезно знать, сколько полей находится в строке. Возможно, вы захотите, чтобы ваш скрипт изменял свою работу в зависимости от количества полей. В качестве примера, команда "ls -l" может генерировать восемь или девять полей, в зависимости от того, какую версию вы выполняете. Версия System V, "/usr/bin/ls -l" генерирует девять полей, что эквивалентно команде Berkeley "/usr/ucb/ls -lg". Если вы хотите напечатать владельца и имя файла, тогда следующий сценарий AWK будет работать с любой версией "ls":

```
#!/bin/awk -f
# проанализируйте вывод "ls -l"
# печать владельца и имени файла
# помните - Berkeley ls -l имеет 8 полей, System V имеет 9
{
 если (NF == 8) {
вывести $ 3, $ 8;
 } еще, если (NF == 9) {
вывести $ 3, $ 9;
 }
}
```

Нажмите здесь, чтобы получить файл: owner_group.awk

Не забывайте, что перед переменной можно добавить символ "$". Это позволяет печатать последнее поле любого столбца

```
#!/bin/awk -f
{ print $NF; }
```

Нажмите здесь, чтобы получить файл: print_last_field.awk

Одно предупреждение о AWK. Существует ограничение в 99 полей в одной строке. У PERL нет таких ограничений.

NR - переменная количества записей

Еще одна полезная переменная - "NR". Здесь указывается количество записей или номер строки. Вы можете использовать AWK только для проверки определенных строк. В этом примере выводятся строки после первых 100 строк и перед каждой строкой после 100 ставится номер строки:

```
#!/bin/awk -f
{ if (NR > 100) {
 распечатать NR, $ 0;
}
```

Нажмите здесь, чтобы получить файл: awk_example5.awk

RS - переменная-разделитель записей

Обычно AWK считывает по одной строке за раз и разбивает строку на поля. Вы можете установить переменную "RS", чтобы изменить определение AWK для "строки". Если вы установите для него пустую строку, то AWK прочитает весь файл в память. Вы можете объединить это с изменением переменной "FS". В этом примере каждая строка обрабатывается как поле, а вторая и третья строки выводятся на печать:

```
#!/bin/awk -f
НАЧАТЬ {
# измените разделитель записей с новой строки на ничто
 RS =""
# измените разделитель полей с пробела на новую строку
 FS="\n"
}
{
# выведите вторую и третью строки файла
print $ 2, $ 3;
}
```

Нажмите здесь, чтобы получить файл: awk_example6.awk

Две строки печатаются с пробелом между ними. Также это будет работать, только если входной файл меньше 100 строк, поэтому этот метод ограничен. Вы можете использовать его для разбиения слов, по одному слову в строке, используя это:

```
#!/bin/awk -f
НАЧАТЬ {
 RS =" ";
}
{
 печать ;
}
```

Нажмите здесь, чтобы получить файл: oneword_per_line.awk

но это работает только в том случае, если все слова разделены пробелом. Если внутри есть табуляция или знаки препинания, этого не будет.

ORS - переменная разделителя выходных записей

Разделителем выходных записей по умолчанию является символ новой строки, как и при вводе. Есл вам нужно сгенерировать текстовый файл для системы, отличной от UNIX, его можно задать как символ новой строки и возврата каретки.

```
#!/bin/awk -f
# этот фильтр добавляет возврат каретки ко всем строкам
# перед символом новой строки
НАЧАТЬ {
 ORS="\r \ n"
}
{ печать }
```

Нажмите здесь, чтобы получить файл: add_cr.awk

FILENAME - текущая переменная Filename

Последняя переменная, известная обычному AWK, - это "FILENAME", которая сообщает вам имя считываемого файла.

```
#!/bin/awk -f
# сообщает, какой файл читается
НАЧАТЬ {
 f ="";
}
{ if (f != FILENAME) {
 выведите "чтение", ИМЯ ФАЙЛА;
 f=ИМЯ ФАЙЛА;
 }
печать;
}
```

Нажмите здесь, чтобы получить файл: awk_example6a.awk

Это можно использовать, если AWK необходимо проанализировать несколько файлов. Обычно вы используете стандартный ввод для предоставления AWK информации. Вы также можете указать имена файлов в командной строке. Если приведенный выше сценарий назывался "testfilter", и если вы выполнили его с

```
тестовый фильтр file1 file2 file3
```

Он будет распечатывать имя файла перед каждым изменением. Альтернативный способ указать это в командной строке

```
файл testfilter1 - file3
```

В этом случае второй файл будет называться "-", что является обычным именем для стандартного ввода. Я использовал это, когда хотел поместить некоторую информацию до и после операции фильтрации. Префикс и постфикс помещают специальные данные до и после реальных данных. Проверив имя файла, вы можете проанализировать информацию по-другому. Это также полезно для сообщения о синтаксических ошибках в определенных файлах:

```
#!/bin/awk -f
{
if (NF == 6) {
 # поступай правильно
 } else {
if (FILENAME == "-" ) {
 выведите "СИНТАКСИЧЕСКАЯ ОШИБКА, неправильное количество полей",
"в STDIN, строка #:", NR, "строка: ", $ 0;
 } еще {
 выведите "СИНТАКСИЧЕСКАЯ ОШИБКА, неправильное количество полей",
"Filename: ", FILENAME, "строка # ", NR, "строка: ", $ 0;
 }
 }
}
```

Нажмите здесь, чтобы получить файл: awk_example7.awk

Ассоциативные массивы

Будучи программистом в 1980-х годах, я использовал несколько языков программирования, таких как BASIC, FORTRAN, COBOL, Algol, PL / 1, DG / L, C и Pascal. AWK был первым языком, который я обнаружил, с ассоциативными массивами. (Язык perl был выпущен позже и имел хэш-массивы, что является одним и тем же. Но я буду использовать термин ассоциативные массивы, потому что именно так их описывает руководство по AWK). Этот термин может быть бессмысленным для вас, но поверьте мне, эти массивы бесценны и значительно упрощают программирование. Позвольте мне описать проблему и показать вам, как ассоциативные массивы можно использовать для сокращения времени кодирования, давая вам больше времени для изучения другой глупой проблемы, с которой вы не хотите иметь дело в первую очередь.

Предположим, у вас есть каталог, переполненный файлами, и вы хотите узнать, сколько файлов принадлежит каждому пользователю и, возможно, сколько дискового пространства принадлежит каждому пользователю. Вы действительно хотите кого-то обвинить; трудно сказать, кому принадлежит какой файл. Фильтр, который обрабатывает выходные данные *ls*, будет работать:

```
ls -l | filter
```

Но это не говорит вам, сколько места использует каждый пользователь. Это также не работает для большого дерева каталогов. Для этого требуются команды *find* и *xargs*:

```
Найти. -введите f -print | xargs ls -l | filter
```

Третья колонка "ls" - это имя пользователя. Фильтр должен подсчитывать, сколько раз он видит каждого пользователя. Типичная программа будет иметь массив имен пользователей и другой

массив, который подсчитывает, сколько раз было просмотрено каждое имя пользователя. Индекс дл обоих массивов одинаков; вы используете один массив для поиска индекса, а второй - для отслеживания количества. Я покажу вам один способ сделать это в AWK - неправильный способ:

```
#!/bin/awk -f
# плохой пример программирования в AWK
# подсчитывается, сколько файлов принадлежит каждому пользователю.
НАЧАТЬ {
 number_of_users=0;
}
{
# необходимо убедиться, что вы просматриваете только строки с 8 или более полями
, если (NF>7) {
 пользователь = 0;
# найдите пользователя в нашем списке пользователей
для (i=1; i<=number_of_users; i++) {
# известен ли пользователь?
 если (имя пользователя [i] == $ 3) {
# нашел это - запомните, где находится пользователь
user = i;
 }
 }
если (пользователь == 0) {
# найдено новое
имя пользователя [++number_of_users] = $ 3;
 пользователь=число_о_пользователей;
 }
# увеличить количество
подсчетов [пользователь]++;
 }
}
КОНЕЦ {
 для (i=1; i<=number_of_users; i++) {
 количество печатей [i], имя пользователя [i]
 }
}
```

Нажмите здесь, чтобы получить файл: awk_example8.awk

Я не хочу, чтобы вы **читали** этот сценарий. Я говорил вам, что это неправильный способ сделать это. Если бы вы были программистом на C и не знали AWK, вы, вероятно, использовали бы технику, подобную приведенной выше. Вот та же программа, за исключением этого примера, которая использует ассоциативные массивы AWK. Важно заметить разницу в размере между этими двумя версиями:

```
#!/bin/awk -f
{
 имя пользователя[$3]++;
}
КОНЕЦ {
 для (i в имени пользователя) {
 выведите имя пользователя [i], i;
 }
}
```

Нажмите здесь, чтобы получить файл: count_users0.awk

Это короче, проще и *намного* проще для понимания - как только вы точно поймете, что такое ассоциативный массив. Концепция проста. Вместо того, чтобы использовать число для поиска записи в массиве, используйте *все, что захотите*. Ассоциативный массив в массиве, индекс которого является строкой. Все массивы в AWK являются ассоциативными. В этом случае индекс в массиве является третьим полем команды "ls", которая является именем пользователя. Если пользователь является "bin", основной цикл увеличивает количество пользователей за счет эффективного выполнения

имя пользователя ["bin"]++;

Гуру UNIX могут радостно сообщить, что 8-строчный сценарий AWK может быть заменен:

awk '{print $3}' | sort | uniq -c | sort -nr

Правда, однако это не позволяет подсчитать общее дисковое пространство для каждого пользователя. Нам нужно добавить немного больше интеллекта в сценарий AWK, и нам нужна правильная основа для продолжения. В программе AWK также есть небольшая ошибка. Если вам нужно "быстрое и грязное" решение, вышеприведенное будет в порядке. Если вы хотите сделать его более надежным, вам придется работать в необычных условиях. Если вы предоставите этой программе пустой файл для ввода, вы получите сообщение об ошибке:

awk: имя пользователя не является массивом

Кроме того, если вы передадите ему вывод "ls -l", строка, в которой указано общее количество, увеличит число несуществующих пользователей. Для устранения этой ошибки используются два метода. Первый учитывает только допустимый ввод:

```
#!/bin/awk -f
{
 если (NF>7) {
 имя пользователя[$3]++;
 }
}
КОНЕЦ {
 для (i в имени пользователя) {
 выведите имя пользователя [i], i;
 }
}
```

Нажмите здесь, чтобы получить файл: count_users1.awk
Это устраняет проблему подсчета строки с итогом. Тем не менее, он по-прежнему выдает ошибку при чтении пустого файла в качестве входных данных. Чтобы устранить эту проблему, общепринятым методом является проверка того, что массив всегда существует и имеет специально значение маркера, которое указывает, что запись недопустима. Затем, сообщая о результатах, игнорируйте недопустимую запись.

```
#!/bin/awk -f
НАЧАТЬ {
 имя пользователя [""]= 0;
}
{
 имя пользователя[$3]++;
}
КОНЕЦ {
 для (i в имени пользователя) {
 если (я != "") {
 выведите имя пользователя [i], i;
 }
 }
}
```

Нажмите здесь, чтобы получить файл: count_users2.awk

Это происходит для устранения другой проблемы. Примените эту технику, и вы сделаете свои программы AWK более надежными и простыми для использования другими.

Многомерные массивы

Некоторые люди спрашивают, может ли AWK обрабатывать многомерные массивы. Это может. Однако вы не используете обычные двумерные массивы. Вместо этого вы используете ассоциативные массивы. (Я даже не упоминал, насколько полезны ассоциативные массивы?) Помните, что вы можете поместить что **угодно** в индекс ассоциативного массива. Это требует

другого подхода к решению проблем, но как только вы поймете, вы не сможете жить без этого. Все, что вам нужно сделать, это создать индекс, который объединяет два других индекса. Предположим, вы хотите эффективно выполнить

```
a[1,2] = y;
```

Это недопустимо в AWK. Тем не менее, следующее совершенно нормально:

```
a[1 "," 2] = y;
```

Помните: оператор конкатенации строк AWK - это пробел. Он объединяет три строки в одну строку "1,2". Затем он использует его в качестве индекса в массиве. Это все, что нужно. Существует одна небольшая проблема с ассоциативными массивами, особенно если вы используете команду *for* для вывода каждого элемента: вы не можете контролировать порядок вывода. Вы можете создать алгоритм для генерации индексов в ассоциативный массив и таким образом управлять порядком. Однако это сложно сделать. Поскольку UNIX предоставляет отличную утилиту сортировки, все больше программистов отделяют обработку информации от сортировки. Я покажу вам, что я имею в виду.

Пример использования ассоциативных массивов AWK

Я часто ловлю себя на том, что постоянно использую определенные техники в AWK. Этот пример продемонстрирует эти методы и проиллюстрирует мощь и элегантность AWK. Программа проста и распространена. Диск заполнен. Кто будет обвинен? Я просто надеюсь, что вы используете эту силу с умом. Помните, что вы можете быть тем, кто заполнил диск.

Разрешив свою моральную дилемму, переложив бремя ответственности прямо на ваши плечи, я подробно опишу программу. Я также расскажу о нескольких советах, которые вы найдете полезными в больших программах AWK. Сначала инициализируйте все массивы, используемые в цикле *for* . Для этой цели будет четыре массива. Инициализация проста:

```
u_count[""]= 0;
g_count[""] = 0;
ug_count[""] = 0;
all_count[""] = 0;
```

Второй совет - выбрать соглашение для массивов. Выбор имен массивов и индексов для каждого массива очень важен. В сложной программе может возникнуть путаница при запоминании, какой массив что содержит. Я предлагаю вам четко определить индексы и содержимое каждого массива. Для демонстрации я буду использовать "_count" для указания количества файлов и "_sum" для указания суммы размеров файлов. Кроме того, часть перед "_" указывает индекс, используемый для массива, который будет либо "u" для пользователя, "g" для группы, "ug" для комбинации пользователя и группы, и "все" для общего количества для всех файлов. В других программах я использовал такие имена, как

```
username_to_directory[имя пользователя]= каталог;
```

Следуйте соглашению, подобному этому, и вам будет трудно забыть назначение каждого ассоциативного массива. Даже когда быстрый взлом возвращается, чтобы преследовать вас три года спустя. Я был там.

Третье предложение - убедиться, что ваш ввод приведен в правильной форме. Обычно неплохо быть пессимистичным, но я добавлю простой, но достаточный тест в этом примере.

```
если (NF != 10) {
# игнорировать
} еще {
```

и т.д.

Я разместил предложение о тестировании и ошибках в начале, чтобы остальной код не был загроможден. AWK не имеет пользовательских функций. NAWK, GAWK и PERL делают.

Следующий совет для сложных сценариев AWK - определить имя для каждого используемого поля. этом случае нам нужны пользователь, группа и размер в дисковых блоках. Мы могли бы

использовать размер файла в байтах, но размер блока соответствует блокам на диске, более точно измерение пространства. Блоки диска можно найти с помощью "ls -s". Это добавляет столбец, поэтому имя пользователя становится четвертым столбцом и т. Д. Поэтому сценарий будет содержать:

```
размер = $ 1;
пользователь = $ 4;
группа = $ 5;
```

Это позволит нам легко адаптироваться к изменениям во входных данных. Мы могли бы использовать "$ 1" во всем сценарии, но если бы мы изменили количество полей, что делает параметр "-s", нам пришлось бы менять ссылку на каждое поле. Вы не хотите проходить через сценарий AWK и менять все "$ 1" на "$ 2", а также менять "$ 2" на "$ 3", потому что на самом деле это "$ 1", который вы только что изменили на "$ 2". **Конечно**, это сбивает с толку. Вот почему рекомендуется присваивать полям имена. Я тоже был там.

Затем скрипт AWK подсчитает, сколько раз встречается каждая комбинация пользователей и групп. То есть я собираюсь создать индекс из двух частей, который содержит имя пользователя и имя группы. Это позволит мне подсчитать, сколько раз встречается каждая комбинация пользователя / группы и сколько места на диске используется.

Подумайте об этом: как бы вы рассчитали общее количество только для пользователя или только для группы? Вы могли бы переписать сценарий. Или вы могли бы взять итоговые данные пользователя / группы и суммировать их с помощью второго скрипта.

Вы могли бы это сделать, но это не способ AWK сделать это. Если вам нужно изучить миллион файлов, и для запуска этого скрипта требуется много времени, повторять эту задачу было бы пустой тратой времени. Также неэффективно требовать два сценария, когда один может сделать все. Правильный способ решения этой проблемы - извлечь как можно больше информации за один проход через файлы. Поэтому этот скрипт найдет количество и размер для каждой категории:

```
Каждый пользователь
, каждая группа
, Каждая комбинация пользователей / групп
, все пользователи и группы
```

Вот почему у меня есть 4 массива для подсчета количества файлов. На самом деле мне не нужны 4 массива, так как я могу использовать формат индекса, чтобы определить, какой массив какой. Но на данный момент это облегчает понимание программы. Следующий совет неочевиден, но вы увидите насколько он полезен. Я упоминал, что индексы в массиве могут быть любыми. Если возможно, выберите формат, который позволяет объединять информацию из нескольких массивов. Я понимаю что сейчас это не имеет смысла, но держитесь. Скоро все станет ясно. Я сделаю это, создав универсальный индекс вида

```
<пользователь> <группа>
```

Этот индекс будет использоваться для всех массивов. Между двумя значениями есть пробел. Это охватывает общее количество для комбинации пользователя / группы. Как насчет трех других массивов? Я буду использовать "*", чтобы указать общее количество для всех пользователей или групп. Поэтому индекс для всех файлов будет "* *", в то время как индекс для всех файлов, принадлежащих пользовательскому *демону*, будет "daemon *". Суть скрипта суммирует количество и размер каждого файла, помещая информацию в нужную категорию. Я буду использовать 8 массивов; 4 для размеров файлов и 4 для количества:

```
u_count[пользователь " *"]++;
g_count["*" группа] ++;
ug_count[пользователь " " группа] ++;
all_count["* *"]++;


u_size[пользователь " *"]+= размер;
g_size["*" группа]+= размер;
ug_size[пользователь " " группа]+= размер;
all_size["* *"]+= размер;
```

Как вы увидите, этот конкретный универсальный индекс упростит сортировку. Также важно отсортировать информацию в удобном порядке. Вы можете **попытаться** принудительно задать определенный порядок вывода в AWK, но зачем работать над этим, когда это однострочная команда для *сортировки*? Самое сложное - найти правильный способ сортировки информации. Этот скрипт будет сортировать информацию, используя размер категории в качестве первого поля сортировки. Наибольшее общее количество будет для всех файлов, так что это будет одна из первых строк вывода. Однако для наибольшего числа связей может быть несколько, и необходимо соблюдать осторожность. Вторым полем будет количество файлов. Это поможет разорвать связь. Тем не менее, я хочу, чтобы итоги и промежуточные итоги были перечислены перед отдельными комбинациями пользователей / групп. Третье и четвертое поля будут сгенерированы индексом массива. Это сложная часть, о которой я вас предупреждал. Скрипт выведет одну строку, но утилита *сортировки* этого не узнает. Вместо этого он будет рассматривать его как два поля. Это позволит объединить результаты, и информация из всех 4 массивов будет выглядеть как один массив. Сортировка третьего и четвертого полей будет в порядке словаря, а не числовой, в отличие от первых двух полей. "*" был использован для того, чтобы эти дополнительные поля были перечислены перед комбинацией отдельных пользователей / групп.

Массивы будут напечатаны в следующем формате:

```
для (i в u_count) {
 если (я != "") {
 выведите u_size[i], u_count[i], i;
 }
}
0
```

Я показал вам только один массив, но все четыре печатаются одинаково. В этом суть сценария. Результаты отсортированы, и я преобразовал пространство во вкладку по косметическим соображениям.

Вывод сценария

Я изменил свой каталог на */usr/ucb*, использовал скрипт в этом каталоге. Ниже приведены выходные данные:

```
подсчет размера группы пользователей
3173 81 * *
3173 81 корень *
2973 75 * персонал
2973 75 корневых сотрудников
88 3 * демон
88 3 корневой демон
64 2 * kmem
64 2 корня kmem
48 1 * tty
48 1 корень tty
```

Здесь говорится, что в этом каталоге находится 81 файл, который занимает 3173 дисковых блока. Все файлы принадлежат root. 2973 дисковых блока принадлежат сотрудникам группы. Существует 3 файла с демоном group, который занимает 88 дисковых блоков.

Как вы можете видеть, первая строка информации - это общая информация для всех пользователей и групп. Вторая строка - это промежуточный итог для пользователя "root". Третья строка - это промежуточный итог для группы "персонал". Поэтому порядок сортировки полезен, с промежуточными итогами перед отдельными записями. Вы можете написать простой скрипт AWK или grep для получения информации только от одного пользователя или одной группы, и информация будет легко сортироваться.

Есть только одна проблема. Каталог */usr/ ucb* в моей системе использует только 1849 блоков; по крайней мере, так сообщает *du*. В чем несоответствие? Скрипт **не** понимает жестких ссылок. Это может не быть проблемой на большинстве дисков, потому что многие пользователи не используют жесткие ссылки. Тем не менее, это приводит к неточным результатам. В этом случае программа *vi* также называется *e*, *ex*, *edit*, *view* и 2 другими именами. Программа существует только один раз, но имеет 7 имен. Вы можете сказать, потому что количество ссылок (поле 2) сообщает 7. Это приводит тому, что файл пересчитывается 7 раз, что приводит к неточному итогу. Исправление заключается в

том, чтобы считать несколько ссылок только один раз. Изучение количества ссылок позволит определить, содержит ли файл несколько ссылок. Однако, как вы можете предотвратить повторный подсчет ссылки? Существует простое решение: все эти файлы имеют одинаковый номер *индекса*. Вы можете найти этот номер с помощью опции *-i* для *ls*. Для экономии памяти нам нужно запоминать только индексы файлов, на которые есть несколько ссылок. Это означает, что мы должны добавить еще один столбец к входным данным и перенумеровать все ссылки на поля. Хорошо, что их всего три. Добавить новое поле будет легко, потому что я последовал своему собственному совету.

Окончательный сценарий должен быть простым для понимания. Я использовал варианты этого сотни раз и считаю, что это демонстрирует мощь AWK, а также дает представление о мощной парадигме программирования. AWK решает проблемы такого типа проще, чем большинство языков. Но вы должны использовать AWK правильно.

Примечание - эта версия была написана для коробки Solaris. Вы должны проверить, генерирует ли *ls* правильное количество аргументов. Возможно, потребуется удалить аргумент *-g* и изменить проверку количества файлов. **Обновлено** Я добавил версию Linux ниже - для загрузки.

Ниже приведена полностью рабочая версия программы, которая точно подсчитывает дисковое пространство:

```
#!/bin/sh
найти . -введите f -print | xargs /usr/bin/ls -islg |
awk '
НАЧАТЬ {
# инициализировать все массивы, используемые в цикле for
u_count[""]=0;
 g_count[""]=0;
 ug_count[""]=0;
 all_count[""]=0;
}
{
# подтвердите свой ввод
, если (NF ! = 11) {
# игнорировать
 } еще {
# присвоить имена полей
в индексе = $ 1;
 размер = $ 2;
 количество ссылок = $ 4;
 пользователь = $ 5;
 группа = $ 6;

# должен ли я считать этот файл?

 doit=0;
 если (количество ссылок == 1) {
# только одна копия - посчитай это
, сделай это++;
 } еще {
# жесткая ссылка - учитывается только первая
увиденная [inode]++;
if (seen[inode] == 1) {
doit++;
 }
 }
# если значение doit равно true, то посчитайте файл
if (doit) {

# общее количество за один проход
# использовать имена массивов описания
# используйте индекс массива, который объединяет массивы

# сначала подсчитывается количество файлов

 u_count[пользователь " *"]++;
 g_count["* " группа]++;
 ug_count[пользователь " " группа]++;
 all_count["* *"]++;

# затем общее используемое дисковое пространство
```

```
 u_size[пользователь " *"]+=размер;
 g_size["* " группа]+= размер;
 ug_size[пользователь " " группа]+= размер;
 all_size["* *"]+=размер;
 }
 }
}
КОНЕЦ {
# вывод в форме, которую можно отсортировать
по (i в u_count) {
 если (я != "") {
 выведите u_size[i], u_count[i], i;
 }
 }
 для (i в g_count) {
 если (я != "") {
 выведите g_size[i], g_count[i], i;
 }
 }
для (i в ug_count) {
 если (я != "") {
 выведите ug_size[i], ug_count[i], i;
 }
 }
для (i в all_count) {
 если (я != "") {
 выведите all_size[i], all_count[i], i;
 }
 }
} ' |
# числовая сортировка - сначала самые большие числа
# сначала отсортируйте поля 0 и 1 (сортировка начинается с 0)
# далее следует сортировка по словарю по полям 2 + 3
сортировка +0nr -2 +2d |
# добавить заголовок
(эхо "размер группы пользователей"; cat -) |
# преобразование пробела в табуляцию - делает вывод приятным
# второй набор кавычек содержит один символ табуляции
tr ' ' ' '
# готово - надеюсь, вам понравится
```

Нажмите здесь, чтобы получить файл: count_users3.awk

Помните, когда я сказал, что мне не нужно использовать 4 разных массива? Я могу использовать только один. Это более запутанно, но более кратко

```
#!/bin/sh
найти . -введите f -print | xargs /usr/bin/ls -islg |
awk '
НАЧАТЬ {
# инициализировать все массивы, используемые в for
количество циклов [""]=0;
}
{
# подтвердите свой ввод
, если (NF ! = 11) {
# игнорировать
 } еще {
# присвоить имена полей
в индексе = $ 1;
 размер = $ 2;
 количество ссылок = $ 4;
 пользователь = $ 5;
 группа = $ 6;

# должен ли я считать этот файл?

 doit=0;
 если (количество ссылок == 1) {
# только одна копия - посчитай это
```

```
, сделай это++;
 } еще {
# жесткая ссылка - учитывается только первая
увиденная [inode]++;
if (seen[inode] == 1) {
doit++;
 }
 }
# если значение doit равно true, то посчитайте файл
if (doit) {

# общее количество за один проход
# использовать имена массивов описания
# используйте индекс массива, который объединяет массивы

# сначала подсчитывается количество файлов

 количество [пользователь " *"] ++;
 количество ["* " группа] ++;
 подсчет [пользователь " " группа]++;
 считайте ["* *"]++;

# затем общее используемое дисковое пространство

 размер [пользователь " *"]+= размер;
 размер ["* " группа]+= размер;
 размер [пользователь " " группа] += размер;
 размер ["* *"]+= размер;
 }
 }
}
КОНЕЦ {
# вывод в форме, которую можно отсортировать
по (i в count) {
 если (я != "") {
 размер печати [i], количество [i], я;
 }
 }
} ' |
# числовая сортировка - сначала самые большие числа
# сначала отсортируйте поля 0 и 1 (сортировка начинается с 0)
# далее следует сортировка по словарю по полям 2 + 3
сортировка +0nr -2 +2d |
# добавить заголовок
(эхо "размер группы пользователей"; cat -) |
# преобразование пробела в табуляцию - делает вывод приятным
# второй набор кавычек содержит один символ табуляции
tr ' ' ' '
# готово - надеюсь, вам понравится
```

Нажмите здесь, чтобы получить файл: count_users.awk

Вот версия, которая работает с современными системами Linux, Но предполагает, что у вас правильные имена файлов (без пробелов и т. Д.): count_users_new.awk

Идеальный вывод PRINTF

До сих пор я описывал несколько простых сценариев, которые предоставляют полезную информацию в несколько уродливом формате вывода. Столбцы могут выстраиваться неправильно, и часто бывает трудно найти закономерности или тенденции без этого единства. Чем больше вы будете использовать AWK, тем больше вам захочется четкого и чистого форматирования. Чтобы достичь этого, вы должны освоить функцию *printf*.

PRINTF - форматирование вывода

Printf очень похож на функцию C с тем же именем. У программистов на C не должно возникнуть проблем с использованием функции *printf*.

Printf имеет одну из этих синтаксических форм:

```
printf (формат);
printf (формат, аргументы ...);
printf (формат)> выражение;
printf (формат, аргументы ...)> выражение;
```

Скобки и точка с запятой не являются обязательными. Я использую только первый формат, чтобы он соответствовал другим близлежащим операторам *printf*. Оператор *печати* будет делать то же самое *Printf* показывает свою реальную мощь при использовании команд форматирования.

Первым аргументом функции *printf* является формат. Это строка или переменная, значение которой является строкой. Эта строка, как и все строки, может содержать специальные escape-последовательности для печати управляющих символов.

Escape-последовательности

Символ "\" используется для "экранирования" или обозначения специальных символов. Список этих персонажей приведен в таблице ниже:

Таблица 5 Escape-последовательностей AWK	
Последовательность	**Описание**
\a	ASCII bell (только для NAWK / GAWK)
\ b	Backspace
\ f	Formfeed
\n	Новая строка
\ r	Возврат каретки
\ t	Горизонтальная вкладка
\ v	Вертикальная вкладка (только для NAWK)
\ ddd	Символ (от 1 до 3 восьмеричных цифр) (только для NAWK)
\ xdd	Символ (шестнадцатеричный) (только для NAWK)
\<Любой другой персонаж>	Этот персонаж

Трудно объяснить различия, не будучи многословным. Надеюсь, я приведу достаточно примеров, чтобы продемонстрировать различия.

С помощью NAWK вы можете печатать три символа табуляции, используя эти три разных представления:

```
printf("\t \ 11 \ x9 \ n");
```

Символ табуляции - десятичный 9, восьмеричный 11 или шестнадцатеричный 09. См. справочную страницу ascii (7) для получения дополнительной информации. Аналогично, вы можете напечатать три символа в двойных кавычках (десятичные 34, шестнадцатеричные 22 или восьмеричные 42), используя

```
printf("\" \ x22 \42 \ n");
```

Вы должны заметить разницу между функцией *printf* и функцией *print* . *Печать* завершает строку символом **ORS** и разделяет каждое поле разделителем **OFS**. *Printf* ничего не делает, если вы не укажете действие. Поэтому вы часто будете заканчивать каждую строку символом новой строки "\ n" и вы должны явно указывать разделяющие символы.

Спецификаторы формата

Сила инструкции *printf* заключается в спецификаторах формата, которые всегда начинаются с символа "%". Спецификаторы формата описаны в таблице 6:

Спецификаторы формата таблицы 6 AWK	
Спецификатор	**Значение**
%c	Символ ASCII

%d	Десятичное целое число
%e	Число с плавающей запятой (инженерный формат)
%f	Число с плавающей запятой (формат с фиксированной запятой)
%g	Чем короче e или f, с удалением конечных нулей
%o	Восьмеричный
%s	Строка
%x	Шестнадцатеричный
%%	Буквальный %

Опять же, я быстро расскажу о различиях. Таблица 3 иллюстрирует различия. В первой строке указано, что "printf(%c \ n",100.0)"" печатает "d".

Таблица AWK 7
Пример преобразования формата

Формат	Значение	Результаты
%c	100.0	d
%c	"100.0"	1 (NAWK?)
%c	42	"
%d	100.0	100
%e	100.0	1.000000e+02
%f	100.0	100.000000
%g	100.0	100
%o	100.0	144
%s	100.0	100.0
%s	"13f"	13f
%d	"13f"	0 (AWK)
%d	"13f"	13 (NAWK)
%x	100.0	64

В этой таблице показаны некоторые различия между AWK и NAWK. Когда строка с цифрами и буквами преобразуется в целое число, AWK вернет ноль, в то время как NAWK преобразует как можно больше. Второй пример, отмеченный "NAWK?", вернет "d" в некоторых более ранних версиях NAWK, в то время как более поздние версии вернут "1".

Используя спецификаторы формата, есть еще один способ печати двойных кавычек с помощью NAWK. Это демонстрирует восьмеричное, десятичное и шестнадцатеричное преобразование. Как вы можете видеть, это не симметрично. Десятичные преобразования выполняются по-разному.

```
printf("%s%s%s%c \ n", "\"", "\ x22", "\ 42", 34);
```

Между символом "%" и символом формата может быть четыре необязательных фрагмента информации. Это помогает визуализировать эти поля как:

%<знак><ноль><ширина>.<точность> формат

Я рассмотрю каждый из них отдельно.

Ширина - указание минимального размера поля

Если после "%" стоит число, это указывает минимальное количество символов для печати. Это поле *ширины*. Пробелы добавляются таким образом, чтобы количество печатных символов равнялось этому числу. Обратите внимание, что это минимальный размер поля. Если поле станет большим, оно будет расти, поэтому информация не будет потеряна. Слева добавляются пробелы.

Этот формат позволяет идеально выстраивать столбцы. Рассмотрим следующий формат:

printf("%st%d \ n", s, d);

Если строка "s" длиннее 8 символов, столбцы не будут выстраиваться в линию. Вместо этого используйте

```
printf("%20s%d \ n", s, d);
```

Если длина строки меньше 20 символов, число будет начинаться с 21-го столбца. Если строка слишком длинная, то два поля будут работать вместе, что затруднит чтение. Возможно, вы захотите поместить один пробел между полями, чтобы убедиться, что у вас всегда будет один пробел между полями. Это очень важно, если вы хотите передать выходные данные в другую программу.

Добавление информационных заголовков делает вывод более читаемым. Имейте в виду, что изменение формата данных может затруднить идеальное выравнивание столбцов. Рассмотрим следующий сценарий:

```
#!/usr/bin/awk -f
НАЧАТЬ {
 printf("Номер строки \ n");
}
{
 printf("%10s %6d \ n", $1, $2);
}
```

Нажмите здесь, чтобы получить файл: awk_example9.awk
Было бы неудобно (простите за выбор слов) добавлять новый столбец и сохранять то же выравнивание. Более сложные форматы потребовали бы много проб и ошибок. Вы должны настроить первый *printf*, чтобы он соответствовал второму утверждению *printf*. Я предлагаю

```
#!/usr/bin/awk -f
НАЧАТЬ {
 printf("%10s %6sn", "Строка", "Число");
}
{
 printf("%10s %6d \ n", $1, $2);
}
```

Нажмите здесь, чтобы получить файл: awk_example10.awk

или даже лучше

```
#!/usr/bin/awk -f
НАЧАТЬ {
 format1 ="%10s %6sn";
 format2 ="%10s %6dn";
 printf(format1, "Строка", "Число");
}
{
 printf(формат2, $ 1, $ 2);
}
```

Нажмите здесь, чтобы получить файл: awk_example11.awk

В последнем примере использование строковых переменных для форматирования позволяет сохранить все форматы вместе. Может показаться, что это не очень полезно, но когда у вас есть несколько форматов и несколько столбцов, очень полезно иметь набор шаблонов, подобных приведенному выше. Если вам нужно добавить дополнительное пространство, чтобы все выстроилось в ряд, гораздо проще найти и исправить проблему с помощью набора строк формата, которые находятся вместе и имеют одинаковую ширину. Упорядочить первую колонку от 10 символов до 11 легко.

Выравнивание по левому краю

В последнем примере перед каждым полем помещаются пробелы, чтобы убедиться, что соблюдена минимальная ширина поля. Что вы делаете, если вам нужны пробелы справа? Добавьте знак отрицания перед шириной:

```
printf("%-10s %-6d \ n", $ 1, $ 2);
```

При этом печатные символы будут перемещены влево, а справа будут добавлены пробелы.

Значение точности поля

Поле точности, представляющее собой число между десятичной дробью и символом формата, является более сложным. Большинство людей используют его с форматом с плавающей запятой (%f), но, что удивительно, его можно использовать с любым символом формата. В восьмеричном, десятичном или шестнадцатеричном формате указывается минимальное количество символов. Нули добавляются для выполнения этого требования. В форматах %e и %f указывается количество цифр после запятой. %e "e + 00" не входит в точность. Формат %g сочетает в себе характеристики форматов %d и %f. Точность определяет количество отображаемых цифр до и после десятичной точки. Поле точности не влияет на поле %c. Формат %s обладает необычным, но полезным эффектом: он определяет максимальное количество значащих символов для печати.

Если первое число после "%" или после "%-" равно нулю, то система добавляет нули при заполнении. Сюда входят все типы форматов, включая строки и формат символов %c. Это означает что "%010d" и "%.10d" оба добавляют начальные нули, давая минимум 10 цифр. Поэтому формат "%10.10d" является избыточным. В таблице 8 приведены некоторые примеры:

Таблица AWK 8
Примеры сложного форматирования

Формат	Переменная	Результаты
%c	100	"d"
%10c	100	"d"
%010c	100	"000000000d"
%d	10	"10"
%10d	10	" 10"
%10.4d	10.123456789	" 0010"
%10.8d	10.123456789	" 00000010"
%.8d	10.123456789	"00000010"
%010d	10.123456789	"0000000010"
%e	987.1234567890	"9.871235e + 02"
%10.4e	987.1234567890	"9.8712e + 02"
%10.8e	987.1234567890	"9.87123457e+02"
%f	987.1234567890	"987.123457"
%10.4f	987.1234567890	" 987.1235"
%010.4f	987.1234567890	"00987.1235"
%10.8f	987.1234567890	"987.12345679"
%g	987.1234567890	"987.123"
% 10 г	987.1234567890	" 987.123"
% 10,4 г	987.1234567890	" 987.1"
% 010.4г	987.1234567890	"00000987.1"
%.8g	987.1234567890	"987.12346"
%o	987.1234567890	"1733"
%10o	987.1234567890	" 1733"
%010o	987.1234567890	"0000001733"
%.8o	987.1234567890	"00001733"

%s	987.123	"987.123"
%10s	987.123	" 987.123"
%10.4c	987.123	" 987."
%010.8c	987.123	"000987.123"
%x	987.1234567890	"3db"
%10x	987.1234567890	"3db"
%010x	987.1234567890	"00000003db"
%.8x	987.1234567890	"000003db"

Для завершения этого урока по *printf* необходима еще одна тема.

Явный вывод файла

Вместо отправки выходных данных в стандартный вывод, вы можете отправить выходные данные в именованный файл. Формат

```
printf("строка \ n")> "/tmp/file";
```

Вы можете добавить к существующему файлу, используя ">>:"

```
printf("строка \ n") >> "/tmp/file";
```

Как и в оболочке, двойные угловые скобки указывают, что выходные данные **добавляются** в файл, а не **записываются** в пустой файл. Добавление к файлу не удаляет старое содержимое. Однако между AWK и оболочкой есть тонкая разница.

Рассмотрим программу оболочки:

```
#!/bin/sh
в то время как x= $(строка)
y
echo есть $ x >> /tmp / a
, y echo есть $ x > /tmp / b
готово
```

Это позволит считывать стандартный ввод и копировать стандартный ввод в файлы "/ tmp / a" и "/ tmp / b". Файл "/ tmp / a" будет увеличиваться, поскольку информация всегда добавляется к файлу. Однако файл "/tmp / b" будет содержать только одну строку. Это происходит потому, что каждый раз, когда оболочка видит символы ">" или ">>", она открывает файл для записи, выбирая при этом опцию усечения / создания или добавления.

Теперь рассмотрим эквивалентную программу AWK:

```
#!/usr/bin/awk -f
{
 выведите $ 0 >> "/ tmp / a"
 вывести $ 0 > "/ tmp / b"
}
```

Это ведет себя по-разному. AWK выбирает опцию создать / добавить при первом открытии файла для записи. После этого использование ">" или ">>" игнорируется. В отличие от оболочки, AWK копирует все стандартные входные данные в файл "/tmp / b".

Вместо строки некоторые версии AWK позволяют указывать выражение:

```
# [примечание для себя] проверьте это - это может не сработать
printf("string \ n")> FILENAME ".out";
```

Ниже используется выражение конкатенации строк, чтобы проиллюстрировать это:

```
#!/usr/bin/awk -f
КОНЕЦ {
 для (i=0; i<30; i++) {
 printf("i=%d \ n", i) > "/tmp /a" i;
```

```
 }
}
```

Нажмите здесь, чтобы получить файл: awk_example12.awk
Этот скрипт никогда не завершается, потому что в AWK может быть открыто 10 дополнительных файлов, а в NAWK - 20. Если вы обнаружите, что это проблема, загляните в PERL.

Я надеюсь, что это даст вам навыки, позволяющие сделать ваше изображение на выходе AWK идеальным.

Управление потоком с помощью next и exit

Вы можете выйти из скрипта awk с помощью команды exit .

```
#!/usr/bin/awk -f
{
 # здесь много кода, где вы можете найти ошибку
, если ( numberOfErrors> 0) {
 выход
 }
}
```

Если вы хотите завершить работу с условием ошибки, чтобы вы могли использовать оболочку для разграничения между обычным и ошибочным завершением, вы можете включить целочисленное значение параметра. Допустим, вы ожидаете, что все строки файла будут состоять из 60 символов, и вы хотите использовать программу awk в качестве фильтра для выхода, если количество символо не равно 60. Некоторые примеры кода могут быть

```
#!/usr/bin/awk -f
{
 если (длина ($ 0)> 60) {
 выход 1
 } еще, если (длина ($0) < 60) {
 выход 2
 }
 Печать
}
```

Существует особый случай, если вы используете более новую версию awk, которая может иметь несколько конечных команд. Если одна из конечных команд выполняет команду "exit", другая конечная команда не выполняется.

```
#!/usr/bin/awk -f
{
 # .... некоторый код здесь
}
ЗАВЕРШЕНИЕ { печать "EXIT1";выход}
ЗАВЕРШЕНИЕ { печать "EXIT2"}
```

Из-за команды "exit" будет выполняться только первая конечная команда.

Команда "далее" также изменит поток программы. Это приводит к остановке текущей обработки пространства шаблонов. Программа считывает следующую строку и снова начинает выполнять команды с новой строки.

Числовые функции AWK

В предыдущих уроках я показал, насколько полезен AWK для обработки информации и создания отчетов. Когда вы добавляете несколько функций, AWK становится еще более, ммм, функциональным.

Существует три типа функций: числовые, строковые и все, что осталось. В таблице 9 перечислены все числовые функции:

Таблица AWK 9

Числовые функции		
Имя	**Функция**	**Вариант**
потому что	косинус	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
опыт	Экспонента	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
int	Целое число	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
журнал	Логарифм	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
грех	Синус	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
sqrt	Квадратный корень	ГЛАЗЕТЬ, глазеть, ГЛАЗЕТЬ
atan2	Арктангенс	ГЛАЗЕЙ, ГЛАЗЕЙ
Рэнд	Случайный	ГЛАЗЕЙ, ГЛАЗЕЙ
srand	Семя случайное	ГЛАЗЕЙ, ГЛАЗЕЙ

Тригонометрические функции

О радость. Бьюсь об заклад, что миллионы, если не десятки, моих читателей ждали, когда я расскажу о тригонометрии. Лично я не часто использую тригонометрию на работе, за исключением случаев, когда я ухожу по касательной.

Извините за это. Я не знаю, что на меня нашло. Обычно я не прибегаю к каламбурам. Я напишу заметку для себя, и после того, как я запишу заметку, я попрошу своего босса перерисовать ее.

А теперь прекрати это!Я ненавижу спорить с самим собой. Я всегда проигрываю. Размышления о математике, которую я изучил во 2 году до н.э. (до компьютеров), кажется, вызывают воспоминания о старшей школе, прыщах и (содрогание) временах, которые лучше оставить забытыми. Стресс от воспоминаний о тех днях, должно быть, заставил меня забыть стандарты, которые я обычно устанавливал для себя. Кроме того, никто все равно не ценит тупой юмор, даже если я нахожу острый способ сказать это.

Я лучше быстро сменю тему. Сочетание юмора и компьютеров - очень серьезное дело.

Вот скрипт NAWK, который вычисляет тригонометрические функции для всех степеней между 0 и 360. Это также показывает, почему нет касательной, секущей или косекантной функции. (Они не нужны). Если вы прочитаете сценарий, вы узнаете о некоторых тонких различиях между AWK и NAWK. Все это в тонкой оболочке демонстрации того, почему мы изучали тригонометрию в первую очередь. Чего еще вы можете желать? О, если вам интересно, я написал это в декабре месяце.

```
#!/usr/bin/nawk -f
#
# Немного тригонометрии...
#
# Этот скрипт AWK отображает значения от 0 до 360
# для основных функций тригонометрии
# но сначала - обзор:
#
# (Примечание для редактора - следующая диаграмма предполагает
# шрифт фиксированной ширины, например Courier.
# в противном случае диаграмма выглядит очень глупо, а не слегка глупо)
#
# Предположим, что следующий прямоугольный треугольник
#
# Угол Y
#
# |
# |
# |
# кондиционер
# |
# |
# +------- Угол X
# b
#
# поскольку треугольник представляет собой прямой угол, то
# X+Y=90
#
# Основные тригонометрические функции. Если вы знаете длину
# из 2 сторон и углов вы можете найти длину третьей стороны.
```

```
# Кроме того, если вы знаете длину сторон, вы можете рассчитать
# углы.
#
# Формулы
#
# синус (X) = a / c
# косинус (X) = b / c
# касательная (X) = a / b
#
# взаимные функции
# котангенс (X) = b / a
# секущая (X) = c / b
# cosecant(X) = c / a
#
# Пример 1)
# если угол равен 30, а гипотенуза (c) равна 10, то
# a = синус (30) * 10 = 5
# b = косинус (30) * 10 = 8,66
#
# Второй пример будет более реалистичным:
#
# Предположим, вы ищете рождественскую елку, и
# разговаривая со своей семьей, вы врезаетесь в дерево
# потому что ваша голова была повернута, и ваши дети спорили о том, кто
# собирался нанести первое украшение на дерево.
#
# Когда вы приходите в себя, вы понимаете, что ваши ноги касаются ствола дерева,
# а ваши глаза находятся в 6 футах от основания ваших обмороженных пальцев.
# Считая звезды, которые вращаются вокруг вашей головы, вы также понимаете
# вершина дерева расположена под углом 65 градусов относительно ваших глаз.

# Вы вдруг понимаете, что дерево высотой 12,84 фута! В конце концов,
# касательная (65 градусов) * 6 футов = 12,84 фута

# Хорошо, это нереально. Не многие люди запоминают
# таблица касательных, или может точно оценивать углы.
# Однако я говорил правду о звездах, вращающихся вокруг головы.

#
НАЧАТЬ {
# присвоить значение для числа пи.
 PI = 3,14159;
# выберите номер "Эд Салливан" - действительно, действительно большой
 БОЛЬШОЙ = 999999;
# выберите два формата
# Держите их близко друг к другу, поэтому, когда один столбец становится больше
# другой столбец можно настроить на ту же ширину
fmt1="%7s %8s %8s %8s %10s %10s %10s %10s %10sn";
# распечатайте заголовок каждого столбца
fmt2 ="%7d %8.2f %8.2f %8.2f %10.2f %10.2f %10.2f % 10.2fn".;
# старый AWK хочет обратную косую черту в конце следующей строки
# для продолжения инструкции по печати
# новый AWK позволяет разбить строку на две, после запятой
printf(fmt1, "Градусы", "Радианы", "Косинус", "Синус",
"Тангенс", "Котангенс", "Секущая", "Косекант");

 для (i=0; i<=360;i++) {
# преобразование градусов в радианы
 r = i * (PI / 180);
# в новом AWK обратная косая черта необязательна
# в СТАРОМ AWK они являются обязательными
printf(fmt2, i, r,
# косинус r
cos(r),
# синус r
sin(r),
#
# Я столкнулся с проблемой при делении на ноль.
# Поэтому мне пришлось протестировать этот случай.
#
# старый AWK считает следующую строку слишком сложной
# Я не против добавить обратную косую черту, но переписать
# следующие три строки кажутся бессмысленными для простого урока.
```

```
# Теперь этот скрипт будет работать только с новым AWK - вздох...
# С положительной стороны,
# Мне больше не нужно добавлять эти обратные косые черты
#
# тангенс r
(cos (r) == 0)? БОЛЬШОЙ: грех (r) / cos (r),
# котангенс r
(sin(r) == 0) ? БОЛЬШОЙ: cos (r) / sin (r),
# секущая r
(cos (r) == 0)? БОЛЬШОЙ: 1 / cos (r),
# косекант r
(sin(r) == 0) ? БОЛЬШОЙ: 1 / грех (r));
 }
# поместите здесь exit , чтобы стандартный ввод не требовался.
 выход;
}
```

Нажмите здесь, чтобы получить файл: trigonometry.awk

NAWK также имеет функцию arctangent . Это полезно для некоторых графических работ, таких как

> тангенс дуги (a / b) = угол (в радианах)

Поэтому, если у вас есть местоположения X и Y, арктангенс отношения покажет вам угол. Функция *atan2*() возвращает значение от отрицательного числа *pi* до положительного числа *pi*.

Экспоненты, логарифмы и квадратные корни

Следующий скрипт использует три другие арифметические функции: *log*, *exp* и *sqrt*. Я хотел показать, как их можно использовать вместе, поэтому я разделил логарифм числа на два, что является еще одним способом нахождения квадратного корня. Затем я сравнил значение показател этого нового журнала со встроенной функцией квадратного корня. Затем я вычислил разницу между ними и преобразовал разницу в положительное число.

```
#!/bin/awk -f
# продемонстрировать использование exp(), log() и sqrt в AWK
# например, в чем разница между использованием логарифмов и обычной арифметики
# примечание - exp и log являются естественными логарифмическими функциями, а не базовыми 1
#
НАЧАТЬ {
# о какой ошибке будет сообщено?
 ОШИБКА = 0.000000000001;
# цикл долгое время
для (i=1;i<=2147483647;i++) {
# найти журнал i
logi=log(i);
# что такое квадратный корень из i?
# разделите журнал на 2
logsquareroot=logi / 2;
# преобразовать журнал i обратно
в squareroot=exp(logsquareroot);
# найдите разницу между логарифмическим вычислением
# и встроенное вычисление
diff=sqrt(i)-squareroot;
# сделать разницу положительной
, если (diff < 0) {
diff *=-1;
 }
если (разница> ОШИБКА) {
 printf("%10d, squareroot: %16.8f, ошибка: %16.14f \ n",
i, squareroot, diff);
 }
 }
 выход;
}
```

Нажмите здесь, чтобы получить файл: awk_example13.awk
Зевайте. Этот пример не слишком увлекателен, за исключением тех, кто любит придираться. Ожидайте, что программа достигнет 3 миллионов, прежде чем вы увидите какие-либо ошибки. Скоро я дам вам более захватывающий пример.

Усечение целых чисел

Все версии AWK содержат функцию *int*. Это усекает число, превращая его в целое число. Его можно использовать для округления чисел путем добавления 0,5:

```
printf("округление %8.4f дает %8dn", x, int(x + 0.5));
```

Случайные числа

В NAWK есть функции, которые могут генерировать случайные числа. Функция *rand* возвращает случайное число от 0 до 1. Вот пример, который вычисляет миллион случайных чисел от 0 до 100 и подсчитывает, как часто использовалось каждое число:

```
#!/usr/bin/nawk -f
# в старом AWK нет rand() и srand()
# они есть только в новом AWK
# насколько случайной является случайная функция?
НАЧАТЬ {
# srand();
 i= 0;
 в то время как (i ++<1000000) {
 x=int(rand()*100 + 0,5);
 y[x]++;
 }
для (i=0;i<=100;i++) {
 printf("%dt%d \ n", y[i],i);
 }
 выход;
}
```

Нажмите здесь, чтобы получить файл: random.awk

Если вы выполните этот скрипт несколько раз, вы получите точно такие же результаты. Опытные программисты знают, что генераторы случайных чисел на самом деле не являются случайными, если они не используют специальное оборудование. Эти числа являются псевдослучайными и вычисляются с использованием некоторого алгоритма. Поскольку алгоритм исправлен, числа повторяются, если только числа не заполнены уникальным значением. Это делается с помощью функции *srand* выше, которая закомментирована. Обычно генератору случайных чисел не присваивается специальное начальное значение, пока ошибки в программе не будут устранены. Нет ничего более неприятного, чем ошибка, которая возникает случайным образом. Функции *srand* может быть присвоен аргумент. Если нет, он использует текущее время и день для генерации начального значения для генератора случайных чисел.

Сценарий лотереи

Я обещал более полезный скрипт. Возможно, это то, чего вы ждете. Он считывает два числа и генерирует список случайных чисел. Я называю сценарий "lotto.awk".

```
#!/usr/bin/nawk -f
НАЧАТЬ {
# Предположим, нам нужно 6 случайных чисел от 1 до 36
# Мы могли бы получить эту информацию, прочитав стандартный ввод,
# но в этом примере будет использоваться фиксированный набор параметров.
#
# Сначала инициализируйте начальный
srand();
# Сколько чисел необходимо?
 ЧИСЛО = 6;
# какое минимальное число
```

```
 МИН = 1;
# и максимум?
 МАКСИМАЛЬНОЕ значение = 36;
# Сколько чисел мы найдем? начните с 0
 Число = 0;
 в то время как (число < ЧИСЛО) {
 r=int(((rand() *(1+ MAX-MIN))+MIN));
# видел ли я этот номер раньше?
 if (array[r] == 0) {
# нет, я не
 Число ++;
 array [r]++;
 }
 }

# теперь выведите все числа по порядку
(i= MIN;i<=MAX;i++) {
# отмечен ли он в массиве?
 если (массив [i]) {
# да
printf("%d", i);
 }
 }
 printf("\n");
 выход;
}
```

Нажмите здесь, чтобы получить файл: lotto.awk
Если вы выиграете в лотерею, пришлите мне открытку.

Строковые функции

Помимо числовых функций, существуют два других типа функций: строки и whatchamacallits . Во-первых, список строковых функций:

Таблица AWK 10 строковые функции	
Имя	**Вариант**
индекс (строка, поиск)	AWK, NAWK, GAWK
длина (строка)	AWK, NAWK, GAWK
разделение (строка, массив, разделитель)	AWK, NAWK, GAWK
substr(строка, позиция)	AWK, NAWK, GAWK
substr(строка, позиция, макс)	AWK, NAWK, GAWK
sub(регулярное выражение, замена)	НЕТ, ГЛАЗЕЙ
sub(регулярное выражение, замена, строка)	НЕТ, ГЛАЗЕЙ
gsub(регулярное выражение, замена)	НЕТ, ГЛАЗЕЙ
gsub(регулярное выражение, замена, строка)	НЕТ, ГЛАЗЕЙ
совпадение (строка, регулярное выражение)	НЕТ, ГЛАЗЕЙ
tolower(строка)	GAWK
тауппер (струнный)	GAWK
asort(строка, [d])	GAWK
asorti(строка, [d])	GAWK
gensub(r, s, h [, t])	GAWK
strtonum(строка)	GAWK

Большинство людей сначала используют AWK для выполнения простых вычислений. Ассоциативные массивы и тригонометрические функции - это несколько эзотерические функции, которые новые пользователи воспринимают с рвением заядлого курильщика на фабрике фейерверков. Я подозреваю, что большинство пользователей добавляют некоторые простые строковые функции в свой репертуар, когда хотят добавить немного больше сложности в свои AWK-скрипты. Я надеюсь, что эта колонка даст вам достаточно информации, чтобы вдохновить вас на следующие усилия.

В оригинальном AWK есть четыре строковые функции: *index*(), *length*(), *split*() и *substr*(). Эти функции довольно универсальны.

Функция длины

Что я могу сказать? Функция *length*() вычисляет длину строки. Я часто использую его, чтобы убедиться, что мой ввод правильный. Если вы хотите игнорировать пустые строки, проверьте длину каждой строки перед обработкой с помощью

```
если (длина($0) > 1) {
...
}
```

Вы можете легко использовать его для печати всех строк длиннее определенной длины и т. Д. Следующая команда объединяет все строки короче 80 символов:

```
#!/bin/awk -f
{
 если (длина ($0) < 80) {
 префикс = "";
 для (i = 1; i<(80-длина ($ 0))/2;i++)
 префикс = префикс " ";
печать префикса $ 0;
 } еще {
 Печать;
 }
}
```

Нажмите здесь, чтобы получить файл: center.awk

Индексная функция

Если вы хотите выполнить поиск специального символа, функция *index*() будет искать определенные символы внутри строки. Чтобы найти запятую, код может выглядеть следующим образом:

```
предложение="Это короткое, бессмысленное предложение.";
if (index(предложение, ",")> 0) {
printf("Найдена запятая в позиции \% d \ n", index(предложение,","));
}
```

Функция возвращает положительное значение при нахождении подстроки. Число указывает местоположение подстроки.

Если подстрока состоит из 2 или более символов, все эти символы должны быть найдены в том же порядке для ненулевого возвращаемого значения. Как и функция *length*(), это полезно для проверки правильности входных условий.

Функция Substr

Функция *substr*() может извлекать часть строки.

Есть два способа его использования:

substr(*строка*, *позиция*)

substr(*строка*, *позиция*, *длина*)

где *string* - строка для поиска, *position* - количество символов для начала поиска, а *length* - количество символов для извлечения (по умолчанию 1).

Простой пример - использовать его для поиска или извлечения определенной строки в фиксированном местоположении.

Если вы хотите напечатать столбцы с 40 по 50, используйте

```
#!/bin/awk -f
{печать substr($ 0,40,10)}
```

Если вы хотите выполнить поиск строки "HELLO" в позиции 20. Это можно легко сделать с помощью

```
#!/bin/awk -f
substr($ 0,20,5) == "ПРИВЕТ" {печать}
```

Одним из распространенных способов использования является разделение строки на две части на основе специального символа. Если вы хотите обработать некоторые почтовые адреса, где вы ищете символ "@", и разделить текст на имя пользователя и имя хоста, следующий фрагмент кода может выполнить эту работу:

```
#!/bin/awk -f
{
# поле 1 - это адрес электронной почты - возможно
, если ((x= индекс($1,"@")) > 0) {
 имя пользователя = substr($ 1,1, x-1);
 имя хоста = substr($ 1,x + 1, длина ($ 1));
# вышесказанное совпадает с
# hostname = substr($ 1,x +1);
 printf("имя пользователя = %s, имя хоста = %s \ n", имя пользователя, имя хоста);
 }
}
```

Нажмите здесь, чтобы получить файл: email.awk
Функция *substr*() принимает два или три аргумента. Первый - это строка, второй - позиция. Необязательный третий аргумент - это длина строки для извлечения. Если третий аргумент отсутствует, используется остальная часть строки.

Функция substr может использоваться многими неочевидными способами. В качестве примера, его можно использовать для преобразования прописных букв в строчные.

```
#!/usr/bin/awk -f
# преобразование прописных букв в строчные
НАЧАТЬ {
 LC="abcdefghijklmnopqrstuvwxyz";
 UC="ABCDEFGHIJKLMNOPQRSTUVWXYZ";
}
{
 out ="";
# посмотрите на каждый символ
для (i= 1;i<=длина ($ 0);i++) {
# получить символ для проверки
char=substr($ 0,i,1);
# это заглавная буква?
 j= индекс (UC, char);
 если (j > 0) {
#
выяснил = out substr(LC, j, 1);
 } else {
out = out символ;
 }
 }
 printf("%s \ n", out);
}
```

Нажмите здесь, чтобы получить файл: upper_to_lower.awk

Функция разделения

Другой способ разделить строку - использовать функцию *split*() . Он принимает три аргумента: строку массив и разделитель. Функция возвращает количество найденных фрагментов. Вот пример:

```
#!/usr/bin/awk -f
НАЧАТЬ {
# этот скрипт разбивает предложение на слова, используя
# пробел в качестве символа, разделяющего слова
string="Это строка, не так ли?";
 поиск =" ";
 n= разделение (строка, массив, поиск);
 для (i=1;i<=n;i++) {
 printf("Word[%d]=%s \n", i,массив [i]);
 }
 выход;
}
```

Нажмите здесь, чтобы получить файл: awk_example14.sh

Третий аргумент обычно состоит из одного символа. Если используется более длинная строка, в качестве разделителя используется только первая буква.

Функция GAWK's Tolower и Toupper

GAWK имеет функции *toupper*() и *tolower*() для удобного преобразования регистра. Эти функции принимают строки, поэтому вы можете сократить приведенный выше сценарий до одной строки:

```
#!/usr/local/bin/gawk -f
{
 печать для уменьшения ($ 0);
}
```

Нажмите здесь, чтобы получить файл: upper_to_lower.gawk

Строковые функции NAWK

NAWK (и GAWK) имеют дополнительные строковые функции, которые добавляют примитивную функциональность, подобную SED: *sub*() , *match*() и *gsub*().

Sub() выполняет подстановку строк, как *sed*. Чтобы заменить "старый" на "новый" в строке, используйте

> sub(/ old/, "new", строка)

Если третий аргумент отсутствует, предполагается, что $0 - строка для поиска. Функция возвращает 1, если происходит подстановка, и 0, если нет. Если в первом аргументе не указаны косые черты, предполагается, что первый аргумент является переменной, содержащей регулярное выражение. *Sub*() изменяет только первое вхождение. Функция *gsub*() похожа на опцию **g** в *sed*: преобразуются все вхождения, а не только первое. То есть, если скороговорка встречается более одного раза в строке (или строке), замена будет выполняться один раз для каждого найденного шаблона. Следующий сценарий:

```
#!/usr/bin/nawk -f
НАЧАТЬ {
 string = "Еще один пример примерного предложения";
 шаблон="[Aa] n";
 если (gsub(шаблон, "AN", строка)) {
 printf("Произошла замена: %s \ n", строка);
 }

 выход;
}
```

Нажмите здесь, чтобы получить файл: awk_example15.awk

при выполнении выведите следующее:

> Произошла замена: еще один пример примерного предложения

Как вы можете видеть, шаблон может быть регулярным выражением.

Функция сопоставления

Как показано выше, *sub*() и *gsub*() возвращают положительное значение, если найдено совпадение. Однако у него есть побочный эффект изменения тестируемой строки. Если вы этого не хотите, вы можете скопировать строку в другую переменную и протестировать запасную переменную. NAWK также предоставляет функцию *match*(). Если функция *match*() находит регулярное выражение, она устанавливает две специальные переменные, которые указывают, где начинается и заканчивается регулярное выражение. Вот пример, который делает это:

```
#!/usr/bin/nawk -f
# продемонстрировать функцию сопоставления

НАЧАТЬ {
 регулярное выражение="[a-zA-Z0-9]+";
}
{
 если (совпадение ($ 0, регулярное выражение)) {
# RSTART - это то, с чего начинается шаблон
# RLENGTH - это длина шаблона
до = substr($ 0,1, RSTART-1);
 шаблон = substr($ 0, RSTART, RLENGTH);
 after = substr($ 0, RSTART + RLENGTH);
 printf("%s<%s>%s \ n", до, шаблон, после);
 }
}
```

Нажмите здесь, чтобы получить файл: awk_example16.awk

Наконец, есть функции whatchamacallit. Я мог бы использовать слово "разное", но его слишком сложно написать. Черт возьми, мне все равно пришлось его искать.

Таблица AWK 11 Разные функции	
Имя	**Вариант**
getline	AWK, NAWK, GAWK
getline	НЕТ, ГЛАЗЕЙ
переменная getline	НЕТ, ГЛАЗЕЙ
переменная getline	НЕТ, ГЛАЗЕЙ
"команда" | getline	НЕТ, ГЛАЗЕЙ
"command" | переменная getline	НЕТ, ГЛАЗЕЙ
система (команда)	НЕТ, ГЛАЗЕЙ
закрыть (команда)	НЕТ, ГЛАЗЕЙ
systime()	GAWK
strftime(строка)	GAWK
strftime(строка, временная метка)	GAWK

Системная функция

В NAWK есть функция *system*(), которая может выполнять любую программу. Возвращает статус завершения программы.

```
if (system("/bin/ rm junk")!= 0)
выведите "команда не сработала";
```

Команда может быть строкой, поэтому вы можете динамически создавать команды на основе входных данных. Обратите внимание, что выходные данные не отправляются в программу NAWK. Вы можете отправить его в файл и открыть этот файл для чтения. Однако есть другое решение.

Функция Getline

В AWK есть команда, которая позволяет принудительно вводить новую строку. Это не требует никаких аргументов. Он возвращает 1 в случае успеха, 0, если достигнут конец файла, и -1, если

возникает ошибка. В качестве побочного эффекта строка, содержащая ввод, изменяется. Этот следующий скрипт фильтрует входные данные, и если в конце строки встречается обратная косая черта, он считывает следующую строку, устраняя обратную косую черту, а также необходимость в ней.

```
#!/usr/bin/awk -f
# ищите a в качестве последнего символа.
# если найдено, прочитайте следующую строку и добавьте
{
 строка = 0 долларов;
 в то время как (substr(строка, длина (строка),1) == "\\") {
# отсечь последнюю
строку символов = substr(строка, 1, длина (строка) -1);
 i= getline;
 если (i> 0) {
 строка = строка $ 0;
 } еще {
 printf("отсутствует продолжение в строке %d \ n", NR);
 }
 }
 линия печати;
}
```

Нажмите здесь, чтобы получить файл: awk_example17.awk

Вместо чтения в стандартные переменные, вы можете указать переменную для установки:

```
получить строку
a_line вывести строку a_line;
```

NAWK и GAWK позволяют присваивать функции *getline* необязательное имя файла или строку, содержащую имя файла. Пример примитивного файлового препроцессора, который ищет строки формата

```
#включить имя файла
```

и заменяет эту строку содержимым файла:

```
#!/usr/bin/nawk -f
{
# примитивный препроцессор включения
, если (($ 1 == "#включить") && (NF == 2)) {
# найдено имя файла
filename = $ 2;
 while (i = getline < имя файла) {
 Печать;
 }
 } еще {
 Печать;
 }
}
```

Нажмите здесь, чтобы получить файл: include.nawk

Исходная строка NAWK также может считываться из канала. Если у вас есть программа, которая генерирует одну строку, вы можете использовать

```
"command" | getline;
вывести 0 долларов;
```

или

```
"command" | getline abc;
распечатать abc;
```

Если у вас есть более одной строки, вы можете просмотреть результаты:

```
    while ("команда" | getline) {
     cmd[i++] = $ 0;
    }
    для (i в cmd) {
     printf("%s=%s\n", i, cmd[i]);
    }
```

Одновременно может быть открыт только один канал. Если вы хотите открыть другой канал, вы должны выполнить

```
закрыть ("команда");
```

Это необходимо, даже если достигнут конец файла.

Функция системного времени

Функция *systime*() возвращает текущее время суток как количество секунд, прошедших с полуночи 1 января 1970 года. Это полезно для измерения того, сколько времени требуется для выполнения части вашего кода GAWK.

```
#!/usr/local/bin/gawk -f
# сколько времени требуется, чтобы сделать несколько циклов?
НАЧАТЬ {
 ЦИКЛЫ = 100;
# выполните тест дважды
start=systime();
для (i=0;i
 }
end = systime();
# подсчитайте, сколько времени требуется для выполнения фиктивного теста
do_nothing = end-start;
# теперь выполните тест еще раз с * ВАЖНЫМ* кодом внутри
start=systime();
for (i = 0;i
# Сколько времени это займет?
 while ("date" | getline) {
date = $0;
 }
 закрыть ("дата");
 }
 end = systime();
 newtime = (end - start) - do_nothing;

 если (newtime <= 0) {
printf("%d циклов было недостаточно для тестирования, увеличьте его \ n",
 ЦИКЛЫ);
 выход;
 } еще {
 printf("выполнение циклов %d заняло %6.4f секунд\ n",
 ЦИКЛЫ, новое время);
 printf("Это % 10,8f секунд на цикл \ n",
(новое время) / ЦИКЛЫ);
# поскольку точность часов составляет +/- одну секунду, какова ошибка
printf("точность этого измерения = %6,2f%%\n",
(1/(новое время))*100);
 }
 выход;
}
```

Нажмите здесь, чтобы получить файл: awk_example17.gawk

Функция Strftime

В GAWK есть специальная функция для создания строк на основе текущего времени. Он основан на функции *strftime*(3c). Если вы знакомы с форматами "+" команды *date*(1), у вас есть хорошая фора для понимания того, для чего используется команда *strftime* . Функция *systime*() возвращает текущую дату в секундах. Не очень полезно, если вы хотите создать строку на основе времени. Хотя вы

могли бы преобразовать секунды в дни, месяцы, годы и т. Д., Было бы проще выполнить "date" и преобразовать результаты в строку. (Смотрите предыдущий сценарий для примера). У GAWK есть другое решение, которое устраняет необходимость во внешней программе.

Функция принимает один или два аргумента. Первый аргумент - это строка, указывающая формат. Эта строка содержит обычные и специальные символы. Специальные символы начинаются с обратной косой черты или символа процента. Символы обратной косой черты с префиксом обратной косой черты те же, что я рассматривал ранее. Кроме того, функция *strftime*() определяет десятки комбинаций, все из которых начинаются с "%". В следующей таблице перечислены эти специальные последовательности:

Таблица AWK 12 Форматы strftime GAWK	
%a	Сокращенное название рабочего дня локали
%A	Полное название рабочего дня локали
%b	Сокращенное название месяца в местном языке
%B	Полное название месяца в локальной сети
%c	"Соответствующее" представление даты и времени в языковом стандарте
%d	День месяца в виде десятичного числа (01--31)
%H	Час (24-часовые часы) в виде десятичного числа (00--23)
%Я	Час (12-часовые часы) в виде десятичного числа (01--12)
%j	День года в виде десятичного числа (001--366)
%m	Месяц в виде десятичного числа (01--12)
%M	Минута в виде десятичного числа (00--59)
%p	Эквивалент языка AM / PM
%S	Второе как десятичное число (00--61).
%U	Номер недели в году (воскресенье - первый день недели)
%w	День недели в виде десятичного числа (0--6). Воскресенье - день 0
%W	Номер недели в году (понедельник - первый день недели)
%x	"Подходящее" представление даты в языковом стандарте
%X	"Подходящее" представление времени в языковом стандарте
%y	Год без столетия в виде десятичного числа (00--99)
%Y	Год с веком в виде десятичного числа
%Z	Название или сокращение часового пояса
%%	Буквально%.

В зависимости от вашей операционной системы и установки у вас также могут быть следующие форматы:

Таблица AWK 13 Дополнительные форматы GAWK strftime	
%D	Эквивалентно указанию %m /%d /%y
%e	День месяца, дополненный пробелом, если это только одна цифра
%h	Эквивалент %b, выше
%n	Символ новой строки (ASCII LF)
%r	Эквивалентно указанию %I:%M:%S %p
%R	Эквивалентно указанию %H:%M
%T	Эквивалентно указанию %H:%M:%S
%t	Символ табуляции
%k	Час в виде десятичного числа (0-23)
%l	Час (12-часовые часы) в виде десятичного числа (1-12)
%C	Столетие, как число между 00 и 99
%u	заменяется днем недели в виде десятичного числа [Понедельник == 1]
%V	заменяется номером недели года (с использованием ISO 8601)
%v	Дата в формате виртуальной машины (например, 20 ИЮНЯ 1991 года)

Одним из полезных форматов является

```
strftime("%y_%m_%d_%H_%M_%S")
```

Это создает строку, содержащую год, месяц, день, час, минуту и секунду в формате, который обеспечивает удобную сортировку. Если бы вы запустили это в полдень на Рождество 1994 года, эт сгенерировало бы строку

```
94_12_25_12_00_00
```

Вот эквивалент команды date в GAWK:

```
#! /usr/local/bin/gawk -f
#

НАЧАТЬ {
 format = "%a %b % e %H:%M:%S %Z %Y";
 печать strftime(формат);
}
```

Нажмите здесь, чтобы получить файл: date.gawk

Вы заметите, что в инструкции begin нет команды exit . Если бы я использовал AWK, необходима инструкция exit. В противном случае он никогда не завершится. Если для каждой прочитанной строк не определено действие, NAWK и GAWK не нуждаются в инструкции exit .

Если вы предоставляете второй аргумент функции *strftime*(), она использует этот аргумент в качестве метки времени вместо текущего системного времени. Это полезно для вычисления будущих времен. Следующий скрипт вычисляет время через неделю после текущего времени:

```
#!/usr/local/bin/gawk -f
НАЧАТЬ {
# получить текущее время
ts = systime();
# время исчисляется секундами, поэтому
one_day = 24 * 60 * 60;
 следующая неделя = ts + (7 * one_day);
 format = "%a %b % e %H:%M:%S %Z %Y";
 печать strftime (формат, next_week);
 выход;
}
```

Нажмите здесь, чтобы получить файл: one_week_later.gawk

Определяемые пользователем функции

Наконец, NAWK и GAWK поддерживают пользовательские функции. Эта функция демонстрирует способ печати сообщений об ошибках, включая имя файла и номер строки, если это необходимо:

```
#!/usr/bin/nawk -f
{
 если (NF != 4) {
 ошибка ("Ожидается 4 поля");
 } еще {
 Печать;
 }
}
ошибка функции (сообщение) {
if (FILENAME != "-") {
printf("%s: ", FILENAME) > "/dev/tty";
 }
 printf("строка # %d, %s, строка: %s \ n", NR, сообщение, $ 0) >> "/dev/tty";
}
```

Нажмите здесь, чтобы получить файл: awk_example18.nawk

Шаблоны AWK

В моем первом учебном пособии по AWK я описал оператор AWK как имеющий вид

```
шаблон {команды}
```

До сих пор я использовал только два шаблона: специальные слова *BEGIN* и *END*. Возможны и другие шаблоны, но я их не использовал. Для этого есть несколько причин. Во-первых, эти шаблоны не нужны. Вы можете дублировать их с помощью оператора *if* . Поэтому это "расширенная функция' Шаблоны или, возможно, лучшее слово - условия, как правило, делают программу AWK непонятной для новичка. Вы можете думать о них как о продвинутой теме, которую следует попробовать после ознакомления с основами.

Шаблон или условие - это просто сокращенный тест. Если условие истинно, действие выполняется. Все реляционные тесты можно использовать как шаблон. Команда "head -10", которая печатает первые 10 строк и останавливается, может быть продублирована с помощью

```
{if (NR <= 10) {print}}
```

Изменение оператора *if* на условие сокращает код:

```
Номер <= 10 {печать}
```

Помимо реляционных тестов, вы также можете использовать тесты сдерживания, т. Е. Содержат ли строки регулярные выражения? Печать всех строк, содержащих слово "special", может быть записана как

```
{if ($ 0 ~ /special/) {печать}}
```

или более кратко

```
$ 0 ~ /специальное предложение/ {печать}
```

Этот тип теста настолько распространен, что авторы AWK допускают третий, более короткий формат:

```
/специальный/ {печать}
```

Эти тесты можно комбинировать с командами AND (*&&*) и OR (||), а также с оператором NOT (*!*). Круглые скобки также можно добавить, если вы сомневаетесь или чтобы прояснить свои намерения

Следующее условие выводит строку, если она содержит слово "целое" или столбцы 1 и 2 содержат "часть1" и "часть2" соответственно.

```
($ 0 ~ / целое/) || (($1 ~ / часть1/) && ($2 ~ /часть2/)) {печать}
```

Это можно сократить до

```
/целиком / || $ 1 ~ /часть1/ && $ 2 ~ /часть2/ {печать}
```

Есть один случай, когда добавление круглых скобок причиняет боль. Условие

```
/целиком/ {печать}
```

работает, но

```
(/ целиком/) {печать}
```

нет. Если используются круглые скобки, необходимо явно указать тест:

```
($ 0 ~ / целое) {печать}
```

Неясная ситуация возникает, когда в качестве условия используется простая переменная. Поскольку переменная *NF* определяет количество полей в строке, можно подумать, что утверждение

```
NF {печать}
```

будет печатать все строки с одним из нескольких полей. Это недопустимая команда для AWK, потом что AWK не принимает переменные в качестве условий. Чтобы предотвратить синтаксическую ошибку, мне пришлось изменить ее на

```
NF != 0 {печать}
```

Я ожидал, что NAWK будет работать, но на некоторых системах SunOS он вообще отказывался печатать какие-либо строки. В новых системах Solaris он вел себя правильно. Опять же, изменение его на более длинную форму работало для всех вариантов. GAWK, как и более новая версия NAWK работал правильно. После этого опыта я решил оставить другие, экзотические варианты в покое. Очевидно, что это неизведанная территория. Я мог бы написать скрипт, который печатает первые 2(строк, за исключением случаев, когда было ровно три поля, если только это не строка 10, используя

```
NF == 3 ? NR == 10: NR < 20 { печать }
```

Но я не буду. Неясность, как и каламбуры, часто недооценивается.

Существует еще один распространенный и полезный шаблон, который я еще не описал. Это шаблон, разделенный запятыми. Общий пример имеет вид:

```
/start/,/stop/ {печать}
```

Эта форма определяет в одной строке условие для включения действия и условие для выключения действия. То есть, когда видна строка, содержащая "start", она печатается. Каждая последующая строка также печатается, пока не будет видна строка, содержащая "stop". Это тоже напечатано, но строка после и все последующие строки не печатаются. Это включение и выключение может повторяться много раз. Эквивалентный код, использующий команду *if*, является:

```
{
if ($ 0 ~ /start/) {
 срабатывает = 1;
 }
если (срабатывает) {
 Печать;
 если ($ 0 ~ / стоп /) {
 срабатывает = 0;
 }
 }
}
```

Условия не обязательно должны быть регулярными выражениями. Также можно использовать реляционные тесты. Ниже печатаются все строки от 20 до 40:

```
(Номер ==20), (НОМЕР ==40) {печать}
```

Вы можете комбинировать реляционные и сдерживающие тесты. Ниже печатается каждая строка, пока не будет видна "остановка":

```
(Номер ==1),/стоп/ {печать}
```

Существует еще одна область путаницы в отношении шаблонов: каждый из них независим от других. В сценарии может быть несколько шаблонов; ни один из них не влияет на другие шаблоны. Если выполняется следующий скрипт:

```
NR==10 {print}
(NR==5),(NR==15) {print}
/xxx/ {print}
(NR==1),/NeVerMatchThiS/ {print}
```

и строка 10 входного файла содержит "xxx", она будет напечатана 4 раза, так как каждое условие истинно. Вы можете рассматривать каждое условие как совокупное. Исключением являются особые условия **НАЧАЛА** и **окончания**. В оригинальной AWK у вас может быть только по одному из каждого В NAWK и GAWK у вас может быть несколько НАЧАЛЬНЫХ или КОНЕЧНЫХ действий.

См. Управление потоком с помощью next и exit для получения специального условия с помощью команды "exit" и нескольких операторов END.

Форматирование программ AWK

У многих читателей возникают вопросы о моем стиле программирования в AWK. В частности, они спрашивают меня, почему я включаю такой код:

```
# Распечатать столбец 3
распечатать 3 доллара;
```

когда я мог использовать

```
вывести 3 доллара # вывести столбец 3
```

В конце концов, по их мнению, точка с запятой не нужна, и комментарии не должны начинаться с первого столбца. Это правда. Тем не менее я избегаю этого. Много лет назад, когда я начал писать программы AWK, я был в замешательстве, когда вложение условий было слишком глубоким. Если я переместил сложный оператор *if* внутрь другого оператора *if*, мое выравнивание фигурных скобок стало неправильным. Это может быть очень сложно исправить, особенно с большими скриптами. В настоящее время я использую *emacs* или *eclipse* для форматирования, но в 1980-х у меня не было такой возможности. В то время моим решением было использовать программу *cb*, которая является "улучшителем языка Си". Включив необязательные точки с запятой и начав комментарии в первом столбце строки, я мог бы отправить свой AWK-скрипт через этот фильтр и правильно выровнять вес код.

Переменные среды

Я описал 7 специальных переменных в AWK и кратко упомянул некоторые другие NAWK и GAWK. Ниже приведен полный список:

Таблица AWK 14
Специальные переменные

Переменная	Цель	AWK	NAWK	GAWK
FS	Разделитель полей	ДА	ДА	ДА
NF	Количество полей	ДА	ДА	ДА
RS	Разделитель записей	ДА	ДА	ДА
NR	Количество входных записей	ДА	ДА	ДА
ИМЯ ФАЙЛА	Текущее имя файла	ДА	ДА	ДА
OFS	Разделитель полей вывода	ДА	ДА	ДА
РЕДАКТОРЫ	Разделитель выходных записей	ДА	ДА	ДА
ARGC	Количество аргументов		ДА	ДА
ARGV	Массив аргументов		ДА	ДА
ARGIND	Индекс ARGV текущего файла			ДА
FNR	Введите номер записи		ДА	ДА
OFMT	Формат вывода (по умолчанию "%.6g")		ДА	ДА
ПЕРВЫЙ СТАРТ	Индекс первого символа после совпадения ()		ДА	ДА
ДЛИНА	Длина строки после сопоставления ()		ДА	ДА
ПОДРАЗДЕЛ	Разделитель по умолчанию с несколькими индексами в массиве (по умолчанию "\ 034")		ДА	ДА
ENVIRON	Массив переменных среды			ДА
ИГНОРИРОВАНИЕ	Игнорировать регистр регулярного выражения			ДА
CONVFMT	формат преобразования (по умолчанию: "%.6g")			ДА
ОШИБКА ОТСУТСТВУЕТ	Текущая ошибка после сбоя getline			ДА
ШИРИНА ПОЛЯ	список ширин полей (вместо использования FS)			ДА
BINMODE	Двоичный режим (Windows)			ДА
ВОРСИНКИ	Включение / выключение режима --lint			ДА
PROCINFO	Массив информации о текущей программе AWK			ДА
RT	Рекордный терминатор			ДА
TEXTDOMAIN	Текстовый домен (т.е. локализация) текущей программы AWK			ДА

Поскольку я уже обсуждал многие из них, я расскажу только о тех, которые я пропустил ранее.

ARGC - число или аргументы (NAWK / GAWK)

Переменная **ARGC** задает количество аргументов в командной строке. Он всегда имеет значение один или более, так как он считает свое собственное имя программы в качестве первого аргумента. Если вы указываете имя файла AWK с помощью параметра "-f", оно не учитывается как аргумент. Если вы добавляете переменную в командную строку, используя форму ниже, NAWK не считается аргументом.

```
файл nawk -f.awk x = 17
```

GAWK выполняет, но это потому, что GAWK требует опции "-v" перед каждым заданием:

```
файл gawk -f.awk -v x = 17
```

Переменные GAWK, инициализированные таким образом, не влияют на переменную **ARGC**.

ARGV - массив аргументов (NAWK / GAWK)

Массив **ARGV** - это список аргументов (или файлов), передаваемых в качестве аргументов командной строки.

ARGIND - индекс аргументов (только для GAWK)

ARGIND указывает текущий индекс в массиве **ARGV**, и поэтому **ARGV[ARGIND]** всегда является текущим именем файла. Поэтому

```
ИМЯ ФАЙЛА == ARGV[ARGIND]
```

всегда верно. Он может быть изменен, позволяя вам пропускать файлы и т. Д.

FNR (NAWK / GAWK)

Переменная **FNR** содержит количество прочитанных строк, но сбрасывается для каждого прочитанного файла. Переменная **NR** накапливается для всех прочитанных файлов. Поэтому, если вы выполняете сценарий awk с двумя файлами в качестве аргументов, каждый из которых содержит 10 строк:

```
nawk '{print NR}' файл file2
nawk '{print FNR}' файл file2
```

первая программа будет печатать числа от 1 до 20, в то время как вторая будет печатать числа от 1 до 10 дважды, по одному разу для каждого файла.

OFMT (NAWK / GAWK)

Переменная OFMT задает формат чисел по умолчанию. Значение по умолчанию равно "%.6g".

ПЕРВЫЙ ЗАПУСК, ДЛИНА и соответствие (NAWK / GAWK)

Я уже упоминал переменные **RSTART** и **RLENGTH** . После вызова функции *match*() эти переменные содержат местоположение в строке шаблона поиска. **RLENGTH** содержит длину этого соответствия

SUBSEP - разделитель многомерных массивов (NAWK / GAWK)

Ранее я описал, как вы можете создавать многомерные массивы в AWK. Они создаются путем объединения двух индексов вместе со специальным символом между ними. Если я использую амперсанд в качестве специального символа, я могу получить доступ к значению в местоположении X, Y по ссылке

```
массив [ X "&" Y]
```

В NAWK (и GAWK) встроена эта функция. То есть вы можете указать элемент массива

```
массив [X, Y]
```

Он автоматически создает строку, помещая специальный символ между индексами. Этот символ является непечатаемым символом "034". Вы можете управлять значением этого символа, чтобы убедиться, что ваши строки не содержат один и тот же символ.

ENVIRON - переменные среды (только для GAWK)

Массив **ENVIRON** содержит переменные среды текущего процесса. Вы можете распечатать свой текущий путь поиска с помощью

```
среда печати ["ПУТЬ"]
```

ИГНОРИРОВАНИЕ (только для GAWK)

Переменная **IGNORECASE** обычно равна нулю. Если вы установите для него значение, отличное от нуля, то все совпадения с шаблоном игнорируют регистр. Поэтому следующее эквивалентно "совпадению grep -i":

```
НАЧАТЬ {IGNORECASE=1;}
/сопоставление / {печать}
```

CONVFMT - формат преобразования (только для GAWK)

Переменная CONVFMT используется для указания формата при преобразовании числа в строку. Значение по умолчанию равно "%.6g". Один из способов усечения целых чисел - преобразовать целое число в строку и преобразовать строку в целое число, изменив действия с помощью **CONVFMT**:

```
a = 12;
b = a "";
CONVFMT = "% 2.2f";
c = a "";
```

Переменные **b** и **c** являются строками, но первая будет иметь значение "12.00", а вторая будет иметь значение "12".

ERRNO - системные ошибки (только для GAWK)

Переменная **ERRNO** описывает ошибку в виде строки после сбоя вызова команды **getline**.

FIELDWIDTHS - поля фиксированной ширины (только для GAWK)

Переменная **FIELDWIDTHS** используется при обработке ввода с фиксированной шириной. Если вы хотите прочитать файл, содержащий 3 столбца данных; первый из них имеет ширину 5 символов, второй - 4, а третий - 7, вы можете использовать *substr* для разделения строки на части. Техника, использующая ширину **поля**, будет:

```
BEGIN {FIELDWIDTHS="5 4 7";}
{ printf("Три поля %s %s %s\n", $1, $2, $3);}
```

AWK, NAWK, GAWK или PERL

На этом заканчивается моя серия учебных пособий по AWK и вариантам NAWK и GAWK. Изначально я намеревался избежать широкого освещения NAWK и GAWK. Я сам пользователь PERL и все еще считаю, что PERL стоит изучать, но я не собираюсь описывать этот очень сложный язык. Понимание AWK очень полезно, когда вы начинаете изучать PERL, и часть меня чувствовала, что обучение вас обширным функциям NAWK и GAWK может побудить вас обойти PERL.

Я обнаружил, что углубляюсь в подробности больше, чем планировал, и надеюсь, вы нашли это полезным. Я многое узнал сам, особенно когда обсуждал темы, не затронутые в других книгах. Это напоминает мне о некоторых заключительных советах: если вы не понимаете, как что-то работает, поэкспериментируйте и посмотрите, что вы можете обнаружить. Удачи и счастливого пробуждения..

Другие мои учебные пособия по оболочке Unix можно найти в моем списке учебных пособий по Unix.Другие учебные пособия по оболочкам можно найти на странице Heiner's SHELLdorado и Chris

F. A. Johnson's Unix Shell

А теперь о рекламе....

Спасибо!

Я хотел бы поблагодарить следующих за отзывы:

Ранджит Сингх
Ричард Дженис Бекерт
Кристиан Хаарманн
Куанг Хе
Гюнтер Кнауф
Чей Уэсли
Тодд Эндрюс
Линн Янг
Крис Холл @_cjh
Эрик Стреб дель Торо
Эдуард Лопес @edouard_lopez
Дональд Лайвли
Пит Мэттсон
Стэн Сиринг
Брайан Клоусон

Этот документ был переведен troff2html версии 0.21 22 сентября 2001 года, а затем отредактирован вручную, чтобы сделать его совместимым с: