Безопасность операционных систем

Взаимодействие ОС и пользователей, командные файлы

Виды интерфейсов

- ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора).

- В ранних ОС интерфейс сводился к языку управления заданиями и не требовал интерактивного взаимодействия человека с машиной.
  - Команды языка управления заданиями набивались на перфокарты, а результаты выполнения задания выводились на печатающее устройство.

- Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов:
  - алфавитно-цифровыми (текстовыми);
  - графическими

Текстовый интерфейс

- При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС.
- Обычно командный язык ОС позволяет:
  - запускать и останавливать приложения;
  - выполнять различные операции с каталогами, файлами;
  - получать информацию о состоянии ОС;
  - администрировать систему.
- Основные свойства:
  - Лаконичность
  - Скорость выдачи команд
  - Необходимость изучения команд

Командный интерпретатор

- Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

- Программный модуль ОС, ответственный за чтение, разбор и выполнение команд называют *командным интерпретатором.*

Графический интерфейс

- Ввод команд может быть упрощен и сведен к операциям с графическими примитивами, если операционная система поддерживает графический пользовательский интерфейс.

- При работе с ним пользователь влияет на работу ОС посредством манипулятора, задействуя соответствующие элементы управления или вводя данные с клавиатуры.
  - Простота освоения
  - Наглядность

История стандарта POSIX

- Распространение компанией AT&T Bell Labs исходных кодов UNIX на выгодных условиях привело к появлению множества апологетов системы, среди которых многие предпринимали попытки внести дополнения и усовершенствования в базовую версию ОС.

- Ряд компаний: Digital Equipment, HewlettPackard, Addamax также предприняли усилия в разработке собственных версий ОС на базе оригинальной UNIX.

- В начале 1980-х Институт инженеров по электротехнике и электронике (Institute for Electrical and Electronics Engineers, IEEE) возглавил проект по стандартизации системы UNIX, который с подачи Ричарда Столлмана (Richard Stallman) стал называться Переносимым интерфейсом операционных систем Unix (Portable Operating System Interface for Unix, POSIX).

POSIX в настоящее время

- POSIX – это набор стандартов, описывающих интерфейсы между операционной системой и прикладной программой.
- Стандарт создан для обеспечения совместимости различных UNIX-подобных операционных систем и переносимости прикладных программ на уровне исходного кода, но может быть использован и для не-Unix систем.
- Стандарт включает
  - конкретизирует системный интерфейс на языке C (C++), включая стандарт самого языка,
  - описание системных вызовов в UNIX,
  - описывает утилиты и командную оболочку sh,
  - стандарты регулярных выражений,
  - системные примитивы, которые могут быть использованы при создании прикладной программы

Разделы POSIX

- **Основные определения (Base definitions)** — список основных определений и соглашений, используемых в спецификациях, и список заголовочных файлов языка Си, которые должны быть предоставлены соответствующей стандарту системой.

- **Оболочка и утилиты (Shell and utilities)** — описание утилит и командной оболочки sh, стандарты регулярных выражений.

- **Системные интерфейсы (System interfaces)** — список системных вызовов языка Си.

- **Обоснование (Rationale)** — объяснение принципов, используемых в стандарте.

Полностью совместимы с POSIX

- A/UX
- BSD/OS
- HP-UX
- IBM AIX
- INTEGRITY
- IRIX
- LynxOS
- Mac OS X
- Minix
- MPE/iX
- OpenSolaris
- OpenVMS
- QNX
- RTEMS
- Solaris
- UnixWare
- velOSity
- VxWorks

Не сертифицированы, но большей частью соответствуют

- BeOS
- FreeBSD
- Linux (большинство дистрибутивов)
- NetBSD
- Nucleus RTOS
- OpenBSD
- Sanos
- SkyOS
- Syllable
- VSTa
- Symbian OS (при помощи PIPS)
- DragonFlyBSD

Shell

- Shell - это командная оболочка. Но это не просто промежуточное звено между пользователем и операционной системой, это еще и мощный язык программирования.
- Программы на языке shell называют сценариями, или скриптами.
- Доступно из скриптов:
  - полный набор команд, утилит и программ UNIX;
  - внутренние команды shell - условные операторы, операторы циклов и пр.
- Shell-скрипты применяются при программировании
  - задач администрирования системы
  - задач, которые не требуют для своего создания полновесных языков программирования.

Интерпретатор по умолчанию

- Все консольные команды в POSIX ОС обрабатываются командным интерпретатором.
- Командный интерпретатор является такой же рядовой программой-утилитой, как всякая другая. По умолчанию определяется интерпретатор с именем bash, но может быть использован и любой другой.
- То, какой интерпретатор использовать, определяется при создании нового имени пользователя и зафиксировано в его записи в /etc/passwd. Позже это может быть изменено.
- Работа с командами системы, переменными окружения и другое - могут существенно или в деталях различаться, в зависимости от того, какой конкретно командный интерпретатор используется, и даже от его версии.

Интерпретатор по умолчанию (2)

- Самый широко используемый интерпретатор - bash.
- Название BASH - это аббревиатура от "Bourne-Again Shell" и игра слов от, ставшего уже классикой, "Bourne Shell" Стефена Бурна (Stephen Bourne).
- BASH стал стандартной командной оболочкой de facto для многих разновидностей UNIX.

Зачем изучать?

1. Ускорение решения повседневных администраторских задач.
2. Во время загрузки Linux выполняется целый ряд сценариев из /etc/rc.d, которые настраивают конфигурацию операционной системы и запускают различные сервисы, поэтому очень важно четко понимать эти скрипты и иметь достаточно знаний, чтобы вносить в них какие либо изменения.
3. Shell-скрипты очень хорошо подходят для быстрого создания прототипов сложных приложений, даже не смотря на ограниченный набор языковых конструкций и определенную "медлительность". Такая метода позволяет детально проработать структуру будущего приложения, обнаружить возможные "ловушки" и лишь затем приступить к кодированию на C, C++, Java, или Perl.

Скрипты неприемлемы (1)

- для ресурсоемких задач, особенно когда важна скорость исполнения (поиск, сортировка и т.п.)
- для задач, связанных с выполнением математических вычислений, особенно это касается вычислений с плавающей запятой, вычислений с повышенной точностью, комплексных чисел (для таких задач лучше использовать C++)
- для кросс-платформенного программирования (для этого лучше подходит язык C)
- для сложных приложений, когда структурирование является жизненной необходимостью (контроль за типами переменных, прототипами функций и т.п.)
- для целевых задач, от которых может зависеть успех предприятия. Для проприетарных, "закрытых" программ (скрипты представляют собой исходные тексты программ, доступные для всеобщего обозрения)

Скрипты неприемлемы (2)

- для проектов, содержащих компоненты, очень тесно взаимодействующие между собой.
- для задач, выполняющих огромный объем работ с файлами
- для задач, работающих с многомерными массивами
- когда необходимо работать со структурами данных, такими как связанные списки или деревья
- когда необходимо предоставить графический интерфейс с пользователем (GUI)
- когда необходим прямой доступ к аппаратуре компьютера
- когда необходимо выполнять обмен через порты ввода-вывода или сокеты
- когда необходимо использовать внешние библиотеки

Простой скрипт

- В простейшем случае, скрипт - это ни что иное, как простой список команд системы, записанный в файл.
- Создание скриптов помогает сохранить время и силы, которые тратятся на ввод последовательности команд всякий раз, когда необходимо их выполнить.

sha-bang

- Если файл сценария начинается с последовательности #!, которая в мире UNIX называется sha-bang, то это указывает системе какой интерпретатор следует использовать для исполнения сценария.
- Это может быть командная оболочка (shell), иной интерпретатор или утилита.

```
#!/bin/sh
#!/bin/bash
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/usr/awk -f
```

- /bin/sh -- командный интерпретатор по-умолчанию (bash для Linux-систем)
- Сигнатура должна указывать правильный путь к интерпретатору.
- Сигнатура #! может быть опущена, если не используются специфичные команды.

Запуск сценария

- Запустить сценарий можно командой bash scriptname.
- Более удобный вариант - сделать файл скрипта исполняемым, командой chmod.
  - chmod 555 scriptname (выдача прав на чтение/исполнение любому пользователю в системе
  - chmod +rx scriptname (выдача прав на чтение/исполнение любому пользователю в системе)
  - chmod u+rx scriptname (выдача прав на чтение/исполнение только "владельцу" скрипта)
- Исполняемый скрипт можно запустить командой ./scriptname.
- Поместив скрипт в каталог /usr/local/bin (нужны права root), можно сделать его доступным для себя и других пользователей системы.
- После этого сценарий можно вызвать, просто напечатав название файла в командной строке и нажав клавишу [ENTER].

Комментарии

- Комментарии. Строки, начинающиеся с символа # (за исключением комбинации #!) -- являются комментариями.

```
# Эта строка -- комментарий.
```

- Комментарии могут располагаться и в конце строки с исполняемым кодом.

```
echo "Далее следует комментарий." # Это комментарий.
```

- Комментариям могут предшествовать пробелы (пробел, табуляция).

Конвейер

- | конвейер. Передает вывод предыдущей команды на ввод следующей или на вход командного интерпретатора shell.
- Этот метод часто используется для связывания последовательности команд в единую цепочку.
  - echo ls -l | sh
  - # Передает вывод "echo ls -l" командному интерпретатору shell,
  - #+ тот же результат дает простая команда "ls -l".
  - cat *.lst | sort | uniq
  - # Объединяет все файлы ".lst", сортирует содержимое и удаляет повторяющиеся строки.
- Конвейеры (еще их называют каналами) -- это классический способ взаимодействия процессов, с помощью которого stdout одного процесса перенаправляется на stdin другого.
- Обычно используется совместно с командами вывода, такими как cat или echo, от которых поток данных поступает в "фильтр" (команда, которая на входе получает данные, преобразует их и обрабатывает).

Завершение и код завершения

- Команда **exit** может использоваться для завершения работы сценария, точно так же как и в программах на языке С. Кроме того, она может возвращать некоторое значение, которое может быть проанализировано вызывающим процессом.
- Каждая команда возвращает *код завершения (иногда код завершения называют возвращаемым значением). В случае успеха команда должна возвращать 0, а в случае ошибки --* ненулевое значение, которое, как правило, интерпретируется как код ошибки.
- Команде **exit** можно явно указать код возврата, в виде: **exit *nnn, где nnn -- это код возврата (число в диапазоне 0 - 255).***
- Когда работа сценария завершается командой **exit** без параметров, то код возврата сценария определяется кодом возврата последней исполненной командой.
- После завершения работы сценария, код возврата можно получить, обратившись из командной строки к переменной *$?, т.е. это будет код возврата последней команды,* исполненной в сценарии.

Внутренние команды

- Внутренняя команда -- это команда, которая встроена непосредственно в Bash.
- Команды делаются встроенными либо из соображений производительности - встроенные команды исполняются быстрее, чем внешние, которые, как правило, запускаются в дочернем процессе, либо из-за необходимости прямого доступа к внутренним структурам командного интерпретатора.
- Действие, когда какая либо команда или сама командная оболочка инициирует (порождает) новый подпроцесс, что бы выполнить какую либо работу, называется ветвлением (forking) процесса.
- Новый процесс называется "дочерним" (или "потомком"), а породивший его процесс - "родительским" (или "предком"). В результате и потомок и предок продолжают исполняться одновременно - параллельно друг другу.
- В общем случае, встроенные команды Bash, при исполнении внутри сценария, не порождают новый подпроцесс, в то время как вызов внешних команд, как правило, приводит к созданию нового подпроцесса.
- Внутренние команды могут иметь внешние аналоги. Например, внутренняя команда Bash - echo имеет внешний аналог /bin/echo и их поведение практически идентично.

Пример

```
#!/bin/bash
echo "Эта строка выводится внутренней командой \"echo\"."
/bin/echo "А эта строка выводится внешней командой /bin/echo."
```

Наиболее популярные внутренние команды

- **echo -** выводит (на stdout) выражение или содержимое.

```
echo Hello
echo $a
```

- **printf** -- команда форматированного вывода, расширенный вариант команды echo и ограниченный вариант библиотечной функции printf() в языке C, к тому же синтаксис их несколько отличается друг от друга.

```
printf format-string... parameter...
```

- **read** - "Читает" значение переменной с устройства стандартного ввода -- stdin, в интерактивном режиме это означает клавиатуру.

```
echo -n «Введите значение переменной 'var1': "
# Ключ -n подавляет вывод символа перевода строки.
read var1
# Перед именем переменной отсутствует символ '$'.
echo "var1 = $var1"
```

Классы внутренних команд

- Работа с файловой системой – cd, pwd, dirs, popd, pushd…
- Арифметические выражения – let, expr…
- Управление переменными – eval, set, unset, export, declare, typeset, readonly
- Управление процессами – exit, exec, kill, wait…
- …

Внешние команды

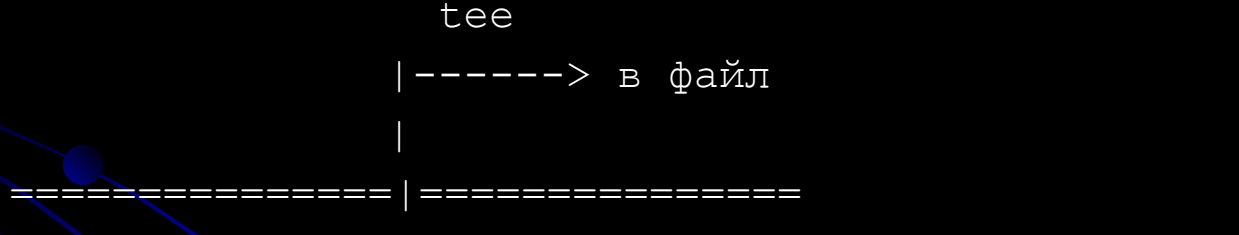
- Работают медленнее внутренних
- Практически неограниченная мощность

Отладка сценариев

- Командная оболочка Bash не имеет своего отладчика, и не имеет даже каких либо отладочных команд или конструкций.
- Синтаксические ошибки или опечатки часто вызывают сообщения об ошибках, которые практически никак не помогают при отладке
- В общих чертах, ошибочными можно считать такие сценарии, которые
  - "сыплют" сообщениями о “синтаксических ошибках”
  - запускаются, но работают не так как ожидалось.
  - запускаются, делают то, что требуется, но имеют побочные эффекты.

Инструменты отладки

- Отладочная печать: команда echo, в критических точках сценария, поможет отследить состояние переменных и отобразить ход исполнения.
- Команда-фильтр tee, которая поможет проверить процессы и потоки данных в критических местах.




```
cat listfile* | sort | tee check.file | uniq > result.file
```

Здесь, в файл check.file будут записаны данные из всех "listfile*", в отсортированном виде до того, как повторяющиеся строки будут удалены командой uniq.

Инструменты отладки

- **ключи -n -v -x**
  - sh -n scriptname -- проверит наличие синтаксических ошибок, не запуская сам сценарий. Того же эффекта можно добиться, вставив в сценарий команду set -n или set -o noexec. Некоторые из синтаксических ошибок не могут быть выявлены таким способом.
  - sh -v scriptname -- выводит каждую команду прежде, чем она будет выполнена. Того же эффекта можно добиться, вставив в сценарий команду set -v или set -o verbose.
  - Ключи -n и -v могут употребляться совместно: sh -nv scriptname.
  - sh -x scriptname -- выводит, в краткой форме, результат исполнения каждой команды. Того же эффекта можно добиться, вставив в сценарий команду set -x или set -o xtrace.
  - Вставка в сценарий set -u или set -o nounset, приводит к получению сообщения об ошибке unbound variable всякий раз, когда будет производиться попытка обращения к необъявленной переменной.

Инструменты отладки

- Функция "assert", предназначенная для проверки переменных или условий, в критических точках сценария. (Эта идея заимствована из языка программирования C.)

```
#!/bin/bash
assert ()     # Если условие ложно,
{       # выход из сценария с сообщением об ошибке.
   E_PARAM_ERR=98
   E_ASSERT_FAILED=99
   if [ -z "$2" ] # Мало входных параметров.
   then
    return $E_PARAM_ERR
   fi
   lineno=$2
   if [ ! $1 ]
   then
    echo "Утверждение ложно: \"$1\""
    echo "Файл: \"$0\", строка: $lineno"
    exit $E_ASSERT_FAILED
   # else
   # return
   # и продолжить исполнение сценария.
   fi
}
```

Инструменты отладки

```
a=5
b=4
condition="$a -lt $b"
   # Сообщение об ошибке и завершение сценария.
   # Попробуйте поменять условие "condition"
   # на что нибудь другое и
   # посмотреть -- что получится.

assert "$condition" $LINENO

   # Сценарий продолжит работу только в том случае, если
   #утверждение истинно.

# Прочие команды.
# ...

echo "Эта строка появится на экране только если
   утверждение истинно."

# ...
# Прочие команды.
# ...
exit 0
```

Инструменты отладки

- Установка ловушек на сигналы

trap 'список команд' сигналы

- Если в системе возникнут прерывания, чьи сигналы перечислены через пробел в "сигналы", то будет выполнен "список команд", после чего (если в списке команд не была выполнена команда "exit") управление вернется в точку прерывания и продолжится выполнение командного файла.
- Например, если перед прекращением по прерываниям выполнения какого то командного файла необходимо удалить файлы в "/tmp", то это может быть выполнено командой "trap" которая предшествует прочим командам файла. Здесь, после удаления файлов будет осуществлён выход "exit" из командного файла.

trap 'rm /tmp/* ; exit 1' 1 2 15

- Команда "trap" позволяет и просто игнорировать прерывания, если "список команд" пустой.

Оптимизация

- По большей части, сценарии на языке командной оболочки, используются для быстрого решения несложных задач. Поэтому оптимизация сценариев, по скорости исполнения, не является насущной проблемой.
- Тем не менее, возможна ситуация, когда сценарий, выполняющий довольно важную работу, в принципе справляется со своей задачей, но делает это очень медленно. Написание же аналогичной программы на языке компилирующего типа - неприемлемо.
- Самое простое решение - переписать самые медленные участки кода сценария.

Советы по оптимизации (1)

- Проверьте все циклы в сценарии. Если это возможно, вынесите все ресурсоемкие операции за пределы циклов.
- Старайтесь использовать встроенные команды. Они исполняются значительно быстрее.
- Избегайте использования избыточных команд, особенно это относится к конвейерам.

```
cat "$file" | grep "$word"
grep "$word" "$file”
```

  - Эти команды дают один и тот же результат, но вторая работает быстрее, поскольку запускает на один подпроцесс меньше.

Советы по оптимизации (2)

- Не следует злоупотреблять командой cat.
- Для профилирования сценариев, можно воспользоваться командами time и times.
- Не следует пренебрегать возможностью переписать особенно критичные участки кода на языке C или даже на ассемблере.
- Попробуйте минимизировать количество операций с файлами. Bash не "страдает" излишней эффективностью при работе с файлами, попробуйте применить специализированные средства для работы с файлами в сценариях, такие как awk или Perl.
- Записывайте сценарии в структурированной форме, это облегчит их последующую реорганизацию и оптимизацию.