mgr Maciej Wróbel

Skrypty Bash, sed, awk itd, zaawansowana obsługa Linux, wyrażenia regularne. Część I.

25 październik 2010

1. Wprowadzenie – skrypty

1.1. Celowość wykorzystania języków skryptowych

Tradycyjny podział na języki niskiego i wysokiego poziomu przez ostatnie lata coraz bardziej ulega zmianie. Te języki, które kiedyś uważane były za języki wysokiego poziomu (jak C) obecnie przez niektórych uważane są za języki niskiego poziomu. Wynika to zarówno z ich bliskiego i silnego związku z konkretną platformą sprzętową, dla której pisane jest oprogramowanie, jak i z dużej prostoty instrukcji, które zamieniane są zwykle na kilka/kilkanaście instrukcji maszynowych. Tymczasem błyskawiczny rozwój oprogramowania powoduje, że programiści chętnie korzystają z gotowych rozwiązań, które pozwalają napisać ten sam program w mniejszej ilości linii kodu. Podejście takie pozwala znacząco skrócić czas tworzenia oprogramowania, a także minimalizuje ilość błędów. Największy mankament takiego podejścia – niższa wydajność aplikacji – coraz rzadziej odgrywa istotną rolę.

Wśród języków najmniej zależnych sprzętowo oraz pozwalających pracować na wysokim poziomie abstrakcji umieszcza się języki skryptowe (takie jak Ruby, Perl, PHP, Python, ECMAScript czy Bash). Pierwotnym ich zastosowaniem było wykonywanie automatycznych czynności oraz uruchamianie i „sklejanie" aplikacji. Obecnie coraz częściej pisze się w nich całe aplikacje. Podstawowe cechy języków skryptowych to: brak typowania zmiennych oraz interpretacja (a nie kompilacja) kodu.

1.2. Bash, Awk i Sed

Z punktu widzenia administracji systemami Linux szczególnie istotna jest znajomość języków wywodzących się z języka sh. Popularnym we wszystkich dystrybucjach jest interpreter Bash (lub Dash, o bardzo podobnej funkcjonalności). Jest to język skryptowy przeznaczony przede wszystkim do pisania prostych programów uruchamiających sekwencje programu, dlatego najczęściej jest wykorzystywany do automatyzowania działań administra-

tora. Język ten posiada większość istotnych konstrukcji, takich jak `if..then..else`, pętle `for, do, while` i definiowanie funkcji. Wraz z standardowo dostępnymi aplikacjami można korzystać w nim z wyrażeń regularnych czy wykonywać obliczenia zmiennoprzecinkowe. W bardzo łatwy sposób można także rozszerzyć jego funkcjonalność pisząc własne programy w językach kompilowanych i wykonując je w kodzie Bash. W języku Bash napisanych jest wiele skryptów startowych systemu i narzędzi administracyjnych. Niektóre dystrybucje, jak np Gentoo posiadają system pakietów (i zarządzania oprogramowaniem) oparty o skrypty w Bash.

Drugim ważnym językiem skryptowym jest język Awk. Pozwala on wykonywać operacje na plikach tekstowych, w oparciu o podział pliku na wiersze, a wierszy na pola. Wykonuje się w nim operacje takie jak sumowanie wartości umieszczonych w kolumnach, zamiana kolejności kolumn itd. lub translacja pomiędzy formatami plików.

Trzecim narzędziem, które wykorzystuje się w administracji systemem (i nie tylko) jest edytor wierszowy Sed. Pozwala on w automatyczny sposób modyfikować wybrane linie plików lub wyrażeń tekstowych, m. in. w oparciu o wyrażenia regularne.

2. Podstawy pisania skryptów w Bash

2.1. Pojęcia wstępne

instrukcja — jest to zwykła nazwa polecenia lub wbudowanej funkcji powłoki, np `ls`, `if`, `seq` itd,

lista instrukcji — jest to... lista instrukcji, rozdzielonych znakami ;, &&, ||[1] lub znakiem nowej linii. Listy instrukcji można grupować operatorami () i {}, które powodują wykonanie sekwencji list instrukcji w nowej podpowłoce (operatory ()) lub wewnątrz aktualnej podpowłoki (zastrzeżone słowa {}, wymagają oddzielenia białymi znakami od znaków tworzących instrukcję).

2.2. Zmienne

W trakcie pracy z Bash można posługiwać się zmiennymi. Zmienne te przechowują łańcuchy znaków ASCII. Zmiennej nadajemy wartość poprzez przypisanie:

```
nazwa_zmiennej='zawartosc zmiennej'
```

Znaki cudzysłowia możemy pominąć, jeżeli wartość składa się z pojedynczego wyrazu. Ponadto, Bash **rozróżnia dwa typy znaków cytowania** i różnie je interpretuje:

1. pojedynczy cudzysłów — obejmujemy nim łańcuchy których znaki nie mają być interpretowane,
2. podwójny cudzysłów — objęte nim łańcuchy traktowane są dosłownie (nie są interpretowane) oprócz znaków `, *, $, !, i \.

   Do wartości zmiennej odwołujemy się poprzedzając jej nazwę znakiem $, np:

[1] Znaczenie operatorów && i || omówione było na poprzednich zajęciach

```
echo $zmienna
```

wyświetli wartość zmiennej *zmienna*. Jeżeli chcemy umieścić wartość zmiennej wewnątrz łańcucha znaków, aby otrzymać jednoznaczne rozwinięcie powinniśmy jej nazwę umieścić w nawiasie klamrowym:

```
zmienna='wartosc'
echo zmienna${zmienna}zmienna
```

wyświetli łańcuch zmiennawartosczmienna. Ważną cechą zmiennych w systemach wywodzących się z UNIX jest to, że są mogą one być wykorzystywane nie tylko przez powłokę, ale także przez skompilowane programy (w C/C++ można ich wartość pobrać funkcjami getenv). Aby zmienna była widoczna w wykonywanych programach musi zostać wyeksportowana poleceniem export:

```
zmienna='abc'
export zmienna
```

lub w jednym wierszu:

```
export zmienna='abc'
```

Nawiasy klamrowe otaczające nazwę zmiennej mają także dodatkowe znaczenie. Jeżeli pojawią się w nich znaki #, %, @, *, :, ?, -, +, =, ^ i znak przecinka, to przed podstawieniem wartości zmiennej zostanie wykonane odpowiednie jej dopasowanie. Szczegóły znajdują się w sekcji Shell Parameter Expansion podręcznika Bash (link). Znaki te pozwalają dokonywać przekształceń wartości zmiennych, jak np.:

```
zmienna=dlugidlugilancuchlancuch
echo ${zmienna%%lancuch*}
echo ${zmienna%*lancuch*}
echo ${zmienna#*dlugi}
echo ${zmienna##*dlugi}
```

zwróci

```
dlugidlugi
dlugidlugilancuch
dlugilancuchlancuch
lancuchlancuch
```

2.3. Podstawianie komend

Ważną umiejętnością jest przekazanie rezultatu działania polecenia do zmiennej środowiskowej. W Bash jest to bardzo proste. Możemy zrobić to na dwa sposoby:

```
zmienna=‘polecenie argumenty‘
```

(znaki ‘to nie apostrofy, a znaki *gravis*, zwykle znajdujące się na tym klawiszu co tylda), lub:

```
zmienna=$(polecenie argumenty)
```

Obydwa wywołania są poprawne i przypiszą do zmiennej to, co wyświetli program *polecenie*, jednak pierwsze z nich nie umożliwia zagnieżdżania wywołań. Przykładem użycia podstawiania komend jest:

```
pliki=‘ls a*|grep b‘;echo $pliki
```

Powyższa lista poleceń wyświetli wszystkie pliki zaczynające się na literę a i zawierające b.

2.4. Dopasowywanie wzorców i podstawianie nazw plików i rozwijanie klamer

Znaki *, ? i [] są traktowane w Bash specjalnie jeżeli nie znajdują się wewnątrz pojedynczego cudzysłowu. Jeżeli Bash znajdzie w łańcuchu znaków któryś z wymienionych znaków, to zakłada, że łańcuch jest wzorcem dopasowania do nazw plików i gdy znajdzie pliki pasujące do wzorca, to zwróci ich listę. W przeciwnym przypadku (domyślnie) pozostawi łańcuch bez zmian. Na przykład:

```
echo a*
```

zwróci listę wszystkich plików zaczynających się na literę a, jeżeli są takie w bieżącym katalogu, a w przeciwnym wypadku wyświetli łańcuch 'a*'. Pliki których nazwy zaczynają się znakiem . (kropką) nie są domyślnie dopasowywane do wzorców (czyli np. ls * nie wyświetli plików o nazwach zaczynających się .).

Wyrażenia ujęte w nawiasy klamrowe { i } rozwijane są podobnie jak nazwy plików, jednak nie muszą istnieć odpowiednie nazwy plików. Przykładowo wyrażenia:

```
a{a,b,c,d}c
a{a..d}c
```

wygenerują łańcuch “aac abc acc adc”. Wyrażenie:

```
a{1..7..2}*
```

wygeneruje łańcuch “a1* a3* a5* a7*”. Ważne jest to, że **rozwijanie klamer wykonywane jest przed pozostałymi podstawieniami**. Dlatego polecenie:

```
echo {a..d}*
```

spowoduje wyświetlenie wszystkich plików o nazwach zaczynających się na a, b, c lub d.

2.5. Arytmetyka powłoki

Wyrażenia objęte znakami $((i)) wykonywane są jako arytmetyczne wyrażenia całkowitoliczbowe. Przykładowo $((10/3)) zwróci wartość 3. Wewnątrz wyrażenia możemy odwoływać się do zmiennych, np.

```
a=3
b=10
echo $(($b/$a))
```

przy czym znaki $ wewnątrz $((,)) można pominąć.

2.6. Wyrażenia logiczne

Wiele operacji administracyjnych wykonuje się warunkowo, w zależności od rezultatu testów logicznych. Bash umożliwia to przez wyrażenia zawarte w nawiasach [] oraz funkcję test:

```
[ wyrazenie_logiczne ]
```

lub

```
test wyrazenie_logiczne
```

Działanie ich jest jednakowe, różnią się tylko składnią (jak widać powyżej). Obydwa zwracają kod wyjścia równy 0, jeżeli wyrażenie logiczne jest prawdzie i różny od zera, jeśli fałszywe. Można testować mi. in wyrażenia logiczne dotyczące:

wyrażeń arytmetycznych:

```
1. test int1 -eq int2
2. test int1 -ne int2
3. test int1 -gt int2
4. test int1 -le int2
...
```

wyrażeń logicznych:

```
1. test  (wyrazenie)
2. test (!wyrazenie)
```

własności łańcuchów:

```
1. test lancuch1 = lancuch2
2. test lancuch1 != lancuch2
3. test -z lancuch1
```

(gdzie 3 przykład zwraca 0, jeżeli łańcuch ma długość 0)

Więcej informacji o poleceniu test na odpowiednich stronach podręcznika (link).

Wyrażenia warunkowe mają zastosowanie w połączeniu z operatorami &&, ||, w pętlach while i until oraz w konstrukcji if..then..else. Na przykład:

```
[ "$zmienna1" = "$zmienna2" ]&&echo ok
```

wyświetli ok, jeżeli zmienna1 jest równa zmiennej 2[2] .

[2] Należy zwracać uwagę na znaki cudzysłowia w wyrażeniach logicznych, ze względu na dzielenie zmiennych na wyrazy. Zaleca się cytowanie podwójnym cudzysłowiem każdej zmiennej użytej w warunku logicznym

2.7. Rurociągi (ang. *pipelines*)

Listy instrukcji w Bash można łączyć operatorami | oraz |&. Powodują one przekierowanie wyjścia standardowego (operator |) lub standardowego wyjścia błędów (operator |&) polecenia po lewej stronie operatora z wejściem standardowym polecenia po prawej stronie. Połączenia takie nazywamy rurociągami (ang. *pipelines*). Ich istnienie jest jednym z najważniejszych funkcjonalności Bash. Dodatkowo możemy przekierować standardowe wejście, wyjście i wyjście błędów operatorami >,< i 2>.

3. Programy w Bash

Wiele prostych skryptów w Bash mieści się w jednej linii kodu. Jeżeli jednak skrypt jest złożony, lub po prostu chcemy mieć możliwość użycia go w przyszłości, dobrze jest mieć możliwość zapisania skryptu jako pliku. Możemy to zrobić, zapisując treść skryptu jako zwykły plik tekstowy. Uruchomić program napisany w Bash i zapisany do pliku można na kilka sposobów. Jeżeli nadamy plikowi z programem w Bash uprawnienia wykonania i go uruchomimy poleceniem:

```
./nazwa_pliku argumenty
```

to wykonany zostanie zapisany tam kod. Jeżeli używaną powłoką jest Bash, to program powinien wykonać się poprawnie. Aby uniezależnić się od używanej powłoki, możemy wywołać program jako argument polecenia Bash:

```
bash nazwa_pliku argumenty
```

Drugim sposobem na uniezależnienie się od powłoki jest dodanie tzw. *shebang line*, czyli:

```
#/bin/bash
```

w pierwszej linijce kodu programu. Spodowuje to, że w razie wykonania program zostanie zinterpretowany poleceniem /bin/bash[3].

Powyższe metody wykonują plik w nowej podpowłoce. Powoduje to, że zmienne ustawiane w programie nie będą ustawione po zakończeniu działania programu. Aby wykonać program w bieżącej podpowłoce (np. po to, aby ustawić zmienne dla całej sesji) wywołujemy poleceniem `source` (którą można także skrócić jako .):

```
source nazwa_pliku argumenty
. nazwa_pliku argumenty
```

3.1. Kolejność wykonywania kodu

W złożonych programach musimy mieć możliwość kontrolowania wykonywania kodu. Listy instrukcji wykonywane są kolejno, aż do zakończenia pliku lub do wystąpienia polecenia `exit`. Część kodu możemy warunkowo wykonać korzystając z operatora if:

[3] Podana metoda nie dotyczy tylko Bash – w linię shebang możemy zaopatrzyć pliki większości interpretowanych języków skryptowych

```
if lista instrukcji0
then
 lista instrukcji1
 ...
 [elif lista instrukcji2
 then
  lista instrukcji3
 ]
else
 lista instrukcji4
fi
```

Jego działanie jest następujące: wykonywana jest lista instrukcji0. Jeżeli kod wyjścia ostatniego polecenia jest równy 0, to wykonywana jest lista instrukcji1, a jeżeli różny od 0, to wykonywana jest kolejno każda instrukcja elif (a może ich być wiele) lista instrukcji3. Jeżeli żadna z instrukcji elif nie zostanie wykonana (ze względu na status zakończenia listy instrukcji2), to wykonana zostanie lista instrukcji4. Wystąpienie wyrażeń `elif` i `else` jest opcjonalne.

Kolejnymi niezbędnymi elementami języka programowania są instrukcje pętli. Dostępnych jest kilka ich odmian.

Pierwszy typ pętli to pętla for ... in :

```
for zmienna in lista
do
   instrukcje do wykonania
done
```

w której zmienna przebiegać będzie po wyrazach zawartych na liście. Jeżeli pominiemy wyrażenie in lista, zmiennej przypisywane będą kolejne argumenty pozycyjne (patrz niżej) wywołania skryptu.

Drugi ważny typ to pętla for((;;))

```
for((wyrazenieStart;warunekEnd;wyrazenieInkrementujace))
do
   instrukcje do wykonania
done
```

gdzie wyrażenieStart ustawia wartości początkowe zmiennych, warunekEnd określa, kiedy zakończyć pętlę a wyrażenieInkrementujące określa, jak mają zmienić się zmienne *po* każdym przebiegu pętli.

Trzeci ważny typ to pętle while/until:

```
while warunek
do
```

```
   instrukcje do wykonania
done
```

(polecenie while można zastąpić poleceniem until). Pętla while(until) jest wykonywana tak długo, dopóki warunek jest prawdziwy(fałszywy). Pętlę while i polecenie read wykorzystuje się, aby skrypt wykonywał działanie na danych ze standardowego wejścia. Np program mycat.sh

```
i=0
while read x
do
i=$((i+1))
echo $i $x
done
```

po wywołaniu ./mycat.sh < /etc/passwd wyświetli ponumerowane linie pliku /etc/passwd.

3.2. Argumenty pozycyjne

Często chcemy przekazać do wykonywanego skryptu argumenty, jak np. nazwę pliku dla którego skrypt ma zostać uruchomiony. Bash umożliwia przekazanie ich w wierszu poleceń, jako argumenty pozycyjne programu. Dostępne są one jako $1, $2, ..., $N. Dodatkowo, zawsze dostępne są zmienne $0, które jest nazwą programu oraz $#, które jest równe ilości zmiennych pozycyjnych.

Do zmiennych pozycyjnych możemy odwoływać się bezpośrednio, przez $1 itd, lub z całej ich listy przez $* lub $@. Przykładowo program

```
for x in $@
do
echo $x
done
```

wyświetli wartości wszystkich zmiennych pozycyjnych.

W wersji elektronicznej instrukcji podane niżej linki są hiperłączami.

Literatura

[1] Materiały MiMUW (j. polski) na `http://wazniak.mimuw.edu.pl/index.php?title=Systemy_operacyjne` dotyczące użytkowania systemu uniksopodobnego.

[2] Każdy podstawowy podręcznik systemu Linux.

[3] Materiały (w j. angielskim) dotyczące certyfikatu LPI udostępniane przez IBM na `http://www.ibm.com/developerworks/linux/library/l-lpic1-v3-map/`

[4] Podstawy Bash (w j. polskim) udostępniane przez społeczność Gentoo na `http://www.gentoo.org/doc/pl/articles/bash-by-example-p1.xml`.

[5] Podręcznik Bash (man Bash lub `http://www.gnu.org/software/bash/manual/bash.html`)

[6] Dla zainteresowanych (j. angielski) — `http://mywiki.wooledge.org/` — wiki poświęcone systemom Unix, wraz z materiałami dot. Bash