

Wprowadzenie do systemu UNIX

Interpretery i skrypty cz. 2.

Celem niniejszej instrukcji jest rozwinięcie opisu zagadnień związanych z tworzeniem skryptów w środowiskach interpreterów *Bash*. Skrypty to programy interpretowane, tworzone w języku programowania interpretera, opartym zazwyczaj na języku C (jak w interpreterach *csh* oraz *tcsh*) lub języku interpreterów z rodziny *sh* (oraz *ksh*, *bsh* i *bash*).

Wykonanie skryptu

O wyborze interpretera, który wykona skrypt, decyduje domyślne ustawienie w systemie. Można jednak na nie wpłynąć dwoma metodami:

1. Ustawienie dyrektywy¹ `#!/bin/bash` na początku pliku tekstowego z programem do wykonania pozwala wskazać ścieżkę interpretera, w tym przypadku *Bash*. Skrypt uruchamiamy następnie, podając ścieżkę dostępu do niego, na przykład: `./script.sh`. Oczywiście, na pliku ze skryptem muszą być ustawione odpowiednie uprawnienia.
2. Uruchomienie skryptu poprzez przekazanie ścieżki do niego do komendy interpretera. Na przykład: `bash ./script.sh` spowoduje wykonanie skryptu za pomocą interpretera *Bash*.

Ta forma może być przydatna do szybkiego debugowania skryptów poprzez użycie parametru `-x`: `bash -x ./script.sh`. Powoduje to wyświetlenie każdej linii z wykonanym podstawieniem zmiennych przed jej wykonaniem. Jest to równoznaczne z użyciem `set -x` na początku skryptu.

Czasami może potrzebne być wykonanie skryptu w **aktualnej instancji** *shella*, zamiast uruchamianie nowej. W takim wypadku korzystamy z komendy `source` lub `.` (kropka). Przykładowe wywołania: `source ./script.sh` lub `./script.sh`. Należy pamiętać, iż w tym wypadku zmiany przeprowadzane w środowisku przez skrypt (np. zmiany wartości zmiennych) będą widoczne w aktualnym *shellu*.

Instrukcje warunkowe i warunki

Środowisko *Bash* dostarcza instrukcji warunkowych i pętli, znanych z klasycznych języków programowania.

DOSTĘPNE SĄ INSTRUKCJE WARUNKOWE:

```
if warunek; then
    instrukcja_1
    instrukcja_2
    instrukcja_3
fi
```

Można tworzyć także bardziej rozbudowane instrukcje warunkowe:

Więcej informacji można znaleźć w *Advanced Bash-Scripting Guide*: <http://www.tldp.org/LDP/abs/html/index.html>

Często popełniane przez początkujących błędy zostały przedstawione tutaj: <http://wiki.bash-hackers.org/scripting/mistakes>

¹ Tak zwany *shebang*.

Dokładniejszy opis wyrażenia `if` znajduje się na stronie: <https://linuxacademy.com/blog/linux/conditions-in-bash-scripting-if-statements/>

```

if warunek_1; then
    instrukcje_jesli_1
elif warunek_2 ; then
    instrukcje_jesli_2_i_nie_1
else
    instrukcje_jesli_nie_2_i_nie_1
fi

```

W miejscu warunek mogą znaleźć się (prawie) dowolne konstrukcje złożone z komend i wyrażeń.

NAJCZĘŚCIEJ SPOTKASZ SIĘ Z WYRAŻENIAMI WARUNKOWYMI zbudowanymi przy użyciu `[[]]` (podwójnych nawiasów kwadratowych). Poniżej przedstawione zostały wybrane operatory:

1. Istnienie plików oraz ich atrybuty:

- `-r PLIK` – `PLIK` istnieje i użytkownik wykonujący test posiada uprawnienia do jego odczytu,
- `-f PLIK` – `PLIK` istnieje i jest plikiem regularnym,
- `-u PLIK` – `PLIK` istnieje i ma ustawioną flagę *SUID*,
- `STARY_PLIK -ot NOWY_PLIK` – `STARY_PLIK` istnieje i jest starszy (zmieniony wcześniej) niż `NOWY_PLIK` albo tylko `NOWY_PLIK` istnieje.

2. Porównywanie ciągów znakowych:

- `STRING_1 == STRING_2` – `STRING_1` jest równy `STRING_2`,
- `STRING_1 != STRING_2` – `STRING_1` jest różny od `STRING_2`,
- `STRING_1 < STRING_2` – `STRING_1` jest wcześniej (według porządku leksykograficznego) od `STRING_2`,
- `-n STRING` – `STRING` jest niepusty,
- `STRING =~ PATTERN` – `STRING` spełnia wyrażenie regularne `PATTERN`.

W wyrażeniach warunkowych można stosować logiczne *and* (`&&`), *or* (`||`) oraz wymuszanie pierwszeństwa operatorów (`(())`). Na przykład:

```
[[ -r ~/.bashrc && ( -n "$V1" || -n "$V2" ) ]]
```

Czasem można spotkać się z użyciem pojedynczych nawiasów kwadratowych `[]` z powyższymi operatorami. Najczęściej będzie to w skryptach, które muszą zachowywać kompatybilność ze starszymi wersjami *Basha* lub innymi interpreterami. Użycie takich warunków jest poprawne, ale trudniejsze – `[` jest – tak naprawdę – komendą *test* i wymaga poprawnego escape'owania argumentów.

LICZBY CAŁKOWITE należy porównywać z użyciem składni podwójnych nawiasów okrągłych (`(())`) (tzw. ewaluacja arytmetyczna). Na przykład:

- `((INT_1 == INT_2))` – liczba `INT_1` jest równa `INT_2`,

- `((INT_1 != INT_2))` – liczba *INT_1* jest różna od *INT_2*,
- `((INT_1 <= INT_2))` – liczba *INT_1* jest mniejsza lub równa *INT_2*,
- `((INT_1 > INT_2))` – liczba *INT_1* jest większa od *INT_2*.

PONADTO WARUNKIEM MOŻE BYĆ dowolna komenda. Na przykład, poniższy fragment kodu wyświetli „Hura”, jeżeli w systemie istnieje proces uruchomiony przez komendę **firefox**:

```
if ps -eo comm | grep -q firefox; then
    echo "Hura"
fi
```

Jak to działa? Każda komenda zwraca kod wyjścia, który można uzyskać za pomocą zmiennej specjalnej `$?`. Spróbuj wykonać następujące linie:

```
ps -eo comm | grep -q firefox
echo $?
```

Jeżeli w systemie istnieje proces o nazwie **firefox**, to druga linia zwróci 0 (poprawne zakończenie). Jeżeli nie, będzie to wartość różna od 0 (błąd programu). Na podstawie tej zwracanej wartości nasza instrukcja warunkowa podejmuje dalszą decyzję o działaniu. 0 traktowane jest jako prawda, a każda inna wartość – jako fałsz.

Pętle

Bash pozwala na wykorzystywanie pętli. Dostępne są pętle **for**, **while** oraz **until**.

Najczęściej spotykać będziesz się z pętlą, która w innych językach często będzie nazywana *foreach*. Iteruje ona po elementach listy za pomocą podanej zmiennej.

```
for ZMIENNA_ITERACYJNA in LISTA
do
    lista_instrukcji
done
```

ZMIENNA_ITERACYJNA jest nazwą zmiennej, którą chcesz użyć, a *LISTA* – listą elementów rozdzielonych białymi znakami.

Druga wersja **for** to klasyczna inkrementacja zmiennej znana z innych języków programowania (np. C++):

```
for (( WART_POZATKOWA ; WARUNEK_KONCA ; ZMIANA ))
do
    lista_instrukcji
done
```

Pętla **while** również nie różni się wiele pod kątem działania od tych znanych z innych języków.

Więcej o pętlach w *Bash* tutaj: https://bash.cyberciti.biz/guide/Chapter_5:_Bash_Loops

```
while TEST
do
    lista_instrukcji
done
```

W przypadku **until** zawartość pętli wykonuje się, dopóki instrukcja warunkowa jest fałszywa.

```
until TEST_KONCA
do
    lista_instrukcji
done
```

Zarówno w przypadku **while**, jak i **until**, *TEST* to dowolny warunek (lub ich grupa), jaki dopuszczalny jest w instrukcji warunkowej.

Przypomnienie: Działania arytmetyczne

Wykonywanie operacji matematycznych z poziomu powłoki jest możliwe dzięki mechanizmowi *arithmetic expansion*. Pozwala ono na wykonanie operacji i zwrócenie jej wyniku. Poniżej przedstawiono proste wyrażenie zwracające wynik dodawania dwóch liczb:

```
echo $(( 5 + 2 ))
```

O mechanizmie *arithmetic expansion* oraz innych pokrewnych można przeczytać tutaj <http://wiki.bash-hackers.org/syntax/expansion/intro>.

Przypomnienie: Podstawianie komend

Przydatnym mechanizmem jest również tzw. *command substitution*. Pozwala ono na wykorzystanie w komendzie wyjścia zwróconego przez inną komendę, wykonaną wcześniej. Istnieją dwie składnie:

1. Bez możliwości zagnieżdżania: komenda `komenda`
Przykład: `sudo chown `id -u` file`
2. Z możliwością zagnieżdżania: komenda `$(komenda)`
Przykład: `sudo chown $(id -u) file`

Grupowanie komend

W razie potrzeby komendy można grupować na dwa sposoby:

1. z użyciem `{ lista_komend }`, które po prostu grupuje komendy,
2. z użyciem `(lista_komend)`, które wykonuje grupę komend w osobnej instancji *shella*.

W każdym przypadku osobne komendy rozdzielamy średnikiem.

Na przykład:

```
(ls /usr; ls /var)
```

Argumenty

Skrypty mogą być uruchamiane z podanymi w wywołaniu argumentami. Na poprzednich zajęciach przedstawione zostały podstawowe metody dostępu do tych argumentów (poprzez zmienne `$i` określające *i*-ty argument, w tym `$0`, która zwraca nazwę wywoływanego skryptu, oraz `$#` i `$*` – zwracające, odpowiednio, liczbę i listę wszystkich argumentów).

Można także korzystać z polecenia **shift**, które powoduje „przesuwanie” argumentów na wcześniejsze pozycje – pierwszy argument zostaje „zapomniany”, drugi staje się pierwszym, trzeci – drugim itd. Pozwala to na uproszczenie przetwarzania podanych argumentów w pętli.

W przykładowym, podanym poniżej skrypcie, następuje wypisanie wszystkich argumentów:

```
#!/bin/bash
# Wypisanie argumentów dla Bash
licznik=1
while [[ $1 ]]
do
    echo "Argument $licznik to $1"
    licznik=$((licznik + 1))
    shift
done
```

Dobrym zwyczajem jest weryfikacja argumentów przed wykonaniem reszty skryptu. Na przykład, jeżeli oczekujemy od użytkownika, że poda nam jeden argument, będący ścieżką do pliku regularnego, powinniśmy zweryfikować to przed wykonaniem innych działań:

```
#!/bin/bash

if (( $# != 1 )); then
    echo "Zła liczba argumentów"
    exit
elif [[ ! -f $1 ]]; then
    echo "Plik nie jest regularny"
    exit
fi

# Dałszy kod
```

Wykorzystana została tu również komenda **exit**, która kończy wykonanie skryptu.

Zadania

1. Zmodyfikuj podany wcześniej skrypt do wypisywania argumentów tak, by w razie braku jakiegokolwiek argumentu w wywołaniu wypisał

stosowny komunikat i zakończył działanie. Uruchom skrypt poprzez wydanie polecenia `bash -vx skrypt.sh` – gdzie `skrypt.sh` to stworzony wcześniej program. Co można zauważyć w komunikatach „debugowania”?

2. Zapoznaj się z działaniem polecenia **trap**. Spróbuj napisać skrypt, który będzie przechwytywał kilka wybranych sygnałów (np. **SIGINT**) i wyświetlał komunikat, jaki sygnał został przechwycony.
3. Napisz skrypt, który przyjmuje trzy argumenty:

- ścieżkę do pliku regularnego, który użytkownik może czytać,
- ścieżkę do katalogu, gdzie użytkownik ma prawo do zapisu,
- liczbę większą od zera.

Podany (argument 1.) plik kopiuje do podanego katalogu (argument 2.) tyle razy, ile wynosi argument 3. Na końcu nazwy każdej kopii dołącz sufix `-kopia-` i numer kopii.

Przykładowe wywołanie:

`./skrypt ala/ma/kota.txt /tmp/moj_katalog 5` powinno skutkować następującą zawartością katalogu `/tmp/moj_katalog`:

```
kota.txt-kopia-1
kota.txt-kopia-2
kota.txt-kopia-3
kota.txt-kopia-4
kota.txt-kopia-5
```

4. Napisz skrypt, który w nieskończonej pętli będzie co 3 sekundy wypisze liczbę aktualnie obecnych procesów w systemie. Zakończenie działania skryptu powinno odbywać się poprzez kombinację klawiszy **Ctrl-c**.
5. Stwórz skrypt, który jako argument wywołania otrzymuje numer UID. Jeśli został on uruchomiony bez argumentu, to pyta o UID użytkownika. Skrypt, korzystając z pliku `/etc/passwd`, wypisuje zawartość 5. kolumny linii opisującej użytkownika o podanym UID. Jeśli jako argument podane zostanie więcej numerów UID, wypisz analogiczną informację dla wszystkich wskazanych użytkowników.
6. Utwórz skrypt, który wypisuje z katalogu, w którym został uruchomiony, nazwy wszystkich plików i katalogów z zaznaczeniem, czy jest to plik regularny, czy katalog. Sam skrypt powinien zostać na takiej liście pominięty. Dla plików wypisz dodatkowo prawa dostępu dla właściciela.
7. Napisz skrypt, który wyświetli prawa dostępu do wskazanego jako argument pliku z punktu widzenia użytkownika, który wywołał skrypt. Jeśli nie został podany żaden argument, skrypt powinien o niego poprosić.
8. Zapoznaj się z poleceniem **dd**. Wykonaj skrypt, który podzieli wskazany plik na kawałki określonej długości. Program powinien przyjmować trzy argumenty:

Przydatne może być polecenie **sleep**.

Wykorzystaj polecenie **read**.

Mogą być potrzebne wyrażenia warunkowe, pozwalające na sprawdzanie uprawnień.

- plik do kopiowania,
- długość (w bajtach) części pliku,
- katalog, do którego zapisywane będą kawałki pliku.

Wykorzystaj polecenie **dd**. Przykładowe rozwiązanie z użyciem tej komendy może zawierać linię:

```
dd conv=noerror if=$1 of=$3/$1.$PASS bs=$2 skip=$((PASS - 1)) count=1 2>/dev/null
```

Użyte oznaczenia:

- PASS – numer aktualnie tworzonego kawałka pliku,
- \$1 – adres pliku źródłowego,
- \$2 – rozmiar kawałka,
- \$3 – adres katalogu docelowego,

conv=noerror powoduje zaprzestanie kopiowania pliku, jeżeli jest ostatni kawałek ma długość mniejszą niż zadeklarowany w wejściu rozmiar kawałka.

Zadanie sprawdzające

Poniższe zadanie wykorzystuje kompleksowo wiedzę z tego laboratorium. Postaraj się wykonać je samodzielnie w domu. Jeżeli masz z nim problemy, przestuduj ponownie materiały źródłowe, rozwiąż wcześniejsze zadania i podejmij kolejną próbę.

Napisz skrypt, który:

1. co 2 sekundy wypisuje listę zalogowanych użytkowników razem z liczbą procesów, które zostały przez nich uruchomione, a w przypadku wciśnięcia przez użytkownika **Ctrl-c** wypisze aktualną datę i zakończy działanie,
2. otrzyma ścieżki do katalogów i dla każdego z nich policzy, ile jest w nich (oraz rekursywnie w podkatalogach) plików regularnych,
3. skopiuje plik, podany jako 1. argument, do wskazanego katalogu docelowego, podanego jako 2. argument (skrypt powinien obsługiwać sytuacje wyjątkowe związane z brakiem dostępu do wskazanego pliku i katalogu i wyświetlać wtedy stosowne komunikaty).

Podsumowanie komend

Na zajęciach przedstawione zostały następujące komendy i składnia:

Grupa	Komenda
Zmienne specjalne	<code>#, \$0, \$*, \$?, \$\$, \$!</code>
Ewaluacja wyrażeń	<code>test, expr, [[...]], [...], ((...))</code>
Obsługa argumentów	<code>shift</code>
Obsługa sygnałów	<code>trap</code>
Instrukcje warunkowe	<code>if ... then ... fi, case ... esac</code>
Pętle for	<code>for ... do ... done, for ((...,...,...)) do ... done</code>
Pętle while	<code>while ... do ... done, until ... do ... done</code>
Grupowanie komend	<code>{ }, (), &&, , ;</code>