Linux i Python w Elektronicznej Sieci #05: Programowanie w bash'u i podstawy systemów operacyjnych

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2021-04-08

1 Bash jako narzędzie programowania

1.1 Zmienne

Określanie typów zmiennych w bashu odbywa się na podstawie wartości znajdującej się w zmiennej. Zasadniczo wszystkie zmienne są napisami, a interpretacja typu ma miejsce przy ich użyciu (a nie przy tworzeniu). Obsługiwane są liczby całkowite oraz napisy, bash nie posiada wbudowanej obsługi liczb zmiennoprzecinkowych.

```
zmiennaA=-91
zmiennaB="qa z"
zmiennaC=98.6 # to będzie traktowane jako napis a nie liczba
```

Zwróć uwagę na brak spacji pomiędzy nazwą zmiennej a znakiem równości w operacji przypisania - jest to wymóg składniowy.

Odwołanie do zmiennej odbywa się z użyciem znaku dolara, po którym występuje nazwa zmiennej. Nazwa może być ujęta w klamry, ale nie musi (jest to przydatne gdy nie chcemy dawać spacji pomiędzy nazwą zmiennej a np. fragmentem napisu). Rozwijaniu ulegają nazwy zmiennych znajdujące się w napisach umieszczonych w podwójnych cudzysłowach.

```
echo  $zmiennaA ${zmiennaA}AA
echo "$zmiennaA ${zmiennaA}AA"
echo '$zmiennaA ${zmiennaA}AA'
```

Jeżeli chcemy aby zmienna była widoczna przez programy uruchamiane z naszej powłoki (w tym przez kolejne instancje bash'a, odpowiedzialne np. za wykonywanie kodu skryptu uruchamianego z pliku) należy ją wyeksportować za pomocą polecenia `export zmienna` (zwróć uwagę na brak dolara w tym miejscu).

1.2 Podstawowe operacje

Aby wykonać działania arytmetyczne należy umieścić je wewnątrz `$(( i ))`

Dodawanie, mnożenie, odejmowanie zapisuje się i działają one tak jak w normalnej matematyce, dzielenie zapisuje się przy pomocy ukośnika i jest ono zawsze dzieleniem całkowitym:

```
a=12; b=3; x=5; y=6

e=$(( ($a + $b) * 4 - $y ))
c=$((  $x / $y ))

echo $e $c $z
```

Zauważ zachowanie przy odwołaniu do niezainicjalizowanej zmiennej z.

Do operacji arytmetycznych może być też jest wykorzystywane polecenie let. Najczęściej jest stosowane do inkrementacji podanej zmiennej, tak jak w poniższym przykładzie.

```
echo $a
let a++
echo $a
```

Zarówno operator podwójnych nawiasów okrągłych jak i komenda let mogą obsługiwać wyrażenia logiczne. Mimo to operacje logiczne najczęściej obsługiwane są komendą `test` lub operatorem `[ ]` wynik zwracany jest jako kod powrotu. Należy zwrócić uwagę na escapowanie odwrotnym ukośnikiem nawiasów i na to że spacje mają znaczenie. Negację realizuje `!`, należy pamiętać jednak że wynikiem negacji dowolnej liczby jest FALSE.

```
[ \( $a -ge 0 -a $b -lt 2 \) -o $z -eq 5 ]; z=$?

echo $z
```

Wartość zmiennej `z` jest wynikiem warunku: `((a większe równe od zera) AND (b mniejsze od dwóch)) OR (z równe 5)`. Bash stosuje logikę odwróconą 0 oznacza prawdę, coś nie zerowego to fałsz.

Jako operacje podstawowe powinniśmy patrzyć także na wykonanie innych programów i pobieranie ich standardowego wyjścia i/lub kodu powrotu. Pobieranie standardowego wyjścia możemy realizować za pomocą ujęcia polecenia w *backquotes* (`` ` ``) lub operatora `$( )` (pozwala on na zagnieżdżanie takich operacji). Natomiast kod powrotu ostatniej komendy znajduje się w zmiennej `$?` (używaliśmy tego już przy obliczaniu wyrażeń logicznych).

```
a=`cat /etc/issuse`
b=$(cat /etc/issuse; cat /etc/resolv.conf)

echo  $a
echo  $b
echo "$b"
```

Zwróć uwagę na różnicę w wypisaniu zmiennej zawierającej znaki nowej linii objętej cudzysłowami i nie objętej nimi.

Bash nie obsługuje liczb zmiennoprzecinkowych, nieobsługiwane operacje można wykonać za pomocą innego programu np:

```
a=`echo 'print(3/2)' | python3`
b=$(echo '3/2' | bc -l)
echo $a $b
```

inne polecenia

Programowanie w bashu w dużej mierze polega na wywoływaniu innych programów (np. takich jak sed, grep, find, awk). Sam bash oferuje jedynie podstawowe konstrukcje składniowe, obsługę zmiennych i pewnych podstawowych operacji na nich.
Na te zewnętrzne polecenia można patrzeć trochę jak na biblioteki w innych językach programowania – komendy gwarantowane przez standard stanowią „bibliotekę standardową" basha, a inne (np. użyty w powyższym przykładzie artmetyki zmiennoprzecinkowej python) stanowią dodatkowe opcjonalne „biblioteki", które pozwalają na łatwiejsze i szybsze rozwiązywanie problemów. W zasadzie podobnie można patrzeć na wywołania zewnętrznych programów w ramach kodu Pythona, C czy innych języków (niekiedy łatwiej jest zrobić np. `system("mv plik nowyplik")` niż zakodować to bezpośrednio w Pythonie czy w C).

1.3 Uruchamianie kodu z pliku

Dłuższe fragmenty kodu bashowego często wygodniej jest pisać w pliku tekstowym niż bezpośrednio w linii poleceń. Plik taki może zostać wykonany przy pomocy polecenia: `./nazwa_pliku` pod warunkiem że ma prawo wykonalności (powinien także zawierać w pierwszą linii komentarz określający program używany do interpretacji tekstowego pliku wykonywalnego, w postaci: `#!/bin/bash`). Może też być wykonany za pomocą wywołania: `bash nazwa_pliku`.

Przydatną alternatywą dla powyższych metod wykonania kodu zawartego w pliku jest włączenie go do aktualnej sesji basha przy pomocy `. ./nazwa_pliku`. W odróżnieniu od poprzednich metod pozwala to na korzystanie z funkcji i zmiennych zdefiniowanych w tym pliku w kolejnych poleceniach.

1.4 Pętle i warunki

1.4.1 Pętla for

W bashu możemy korzystać z kilku wariantów pętli for. Jednym z najczęściej używanych jest przypadek iterowania po plikach[1]:

```
for nazwa in /tmp/* ; do
        echo $nazwa;
done
```

Możliwe jest też iterowanie po wartościach całkowitych zarówno w stylu „shellowym" jak i w stylu C

```
for i in `seq 0 20`; do
        echo $i;
done

for (( i=0 ; $i<=20 ; i++ )) ; do
        echo $i;
done
```

1.4.2 Pętla while

Często używana jest pętla while w połączeniu z instrukcją `read`[2] co umożliwia przetwarzanie jakiegoś wejścia (wyniku komendy lub pliku) linia po linii (także z podziałem linii na słowa):

```
cat /etc/fstab | while read slowo reszta; do
        echo $reszta;
done
```

Powyższa pętla wypisze po kolei wszystkie wiersze pliku `/etc/fstab` przekazanego przez stdin (przy pomocy komendy `cat`)[3] z pominięciem pierwszego słowa (które wczytywane było do zmiennej slowo).

1. Dokładniej: iteracja odbywa się po liście napisów rozdzielanej spacjami - zobacz rezultat `echo /tmp/*`
2. Polecenie `read` można także wykorzystać do wczytania danych podawanych przez użytkownika do jakiejś zmiennej – np. `read -p "wpisz coś >> " xyz` wczyta tekst do zmiennej `xyz`. `read` z opcją `-e` potrafi kożystać z biblioteki readline, jednak np. współdzieli historię z historią basha. Dlatego często wygodniejsze może być zainstalowanie i użycie `rlwrap`, np: `xyz=`rlwrap -H historia.txt -S "wpisz coś >> " head -n1`.
3. Takie rozwiązanie nazywane jest *martwym kotem* i powinno go się unikać. Lepszym rozwiązaniem jest przekazywanie pliku przez przekierowanie strumienia wejściowego przy pomocy `< plik`, który w tym przypadku powinien znaleźć się za kończącym pętle słowem kluczowym `done`.

przekierowania strumieni a zmienne

Przekierowanie standardowego wyjścia na standardowe wejście odbywa się między dwoma różnymi procesami. Zatem w konstrukcjach typu `while read` pętla while uruchamiana może być procesie potomnym obecnej powłoki. Efektem tego jest iż w niektórych przypadkach wykonywane modyfikacje zmiennych wewnątrz takiej pętli nie będą widoczne poza nią.
Przykładem takiej sytuacji jest poniższy kod (polecenie ps dodano aby pokazać utworzneie procesu potomnego powłoki):

```
zm=0; ps -f
cat /etc/fstab | while read x; do
        [ $zm -lt 1 ] && ps -f
        zm=13
done
echo $zm
```

Jednak analogiczny kod w którym następuje przekierowanie z pliku zadziała poprawnie:

```
zm=0; ps -f
while read x; do
        [ $zm -lt 1 ] && ps -f
        zm=13
done < /etc/fstab
echo $zm
```

Słowa domyślnie rozdzielane są przy pomocy dowolnego ciągu spacji lub tabulatorów, separator można zmienić za pomocą zmiennej `IFS`, np:

```
IFS=":"
while read a b c; do echo "$a -- $c"; done < /etc/passwd
unset IFS # przywracamy domyślne zachowanie read poprzez usunięcie zmiennej IFS
```

Należy mieć na uwadze, że cudzysłowa wokół wypisania zmiennej c są istotne – bez nich znak dwókropka mógłby być zmieniony na spacje.

1.4.3 Instrukcja if

Poznane wcześniej obliczanie wartości wyrażeń logicznych najczęściej stosowane jest w instrukcji warunkowej `if`[4].

```
# instruikcja if - else
if [ "$xx" = "kot" -o "$xx" = "pies" ]; then
        echo  "kot lub pies";
elif [ "$xx" = "ryba" ];  then
        echo  "ryba"
else
        echo  "coś innego"
fi
```

Zauważ że spacje wokół i wewnątrz nawiasów kwadratowych przy warunku są istotne składniowo, zawartość nawiasów kwadratowych to tak naprawdę argumenty dla komendy `test`. Oprócz typowych warunków logicznych możemy sprawdzać np. istnienie plików, czy też ich typ (link, katalog, etc). Szczegółowy opis dostępnych warunków które mogą być użyte w tej konstrukcji znajduje się w `man test`.

4. Może być też stosowane np. w pokazanej wcześniej pętli `while`

Jako warunek może wystąpić dowolne polecenie wtedy sprawdzany jest jego kod powrotu 0 oznacza prawdę / zakończenie sukcesem, a wartość nie zerowa fałsz / błąd

```
if grep '^root:' /etc/passwd > /dev/null; then
        echo /etc/passwd zawiera root-a;
fi
```

Istnieje możliwość skróconego zapisu warunków z użyciem łączenia instrukcji przy pomocy && (wykonaj gdy poprzednia zwróciła zero – true) lub || (wykonaj gdy poprzednia zwróciła nie zero – false):

```
[ -f /etc/issuse ] && echo "jest plik /etc/issuse"

grep '^root:' /etc/passwd > /dev/null && echo /etc/passwd zawiera root-a;
```

1.4.4 Instrukcja case

Instrukcja case służy do rozważania wielu przypadków opartych na równości zmiennej z podanymi napisami.

```
case $xx in
        kot | pies)
                echo  "kot lub pies"
                ;;
        ryba)
                echo  "ryba"
                ;;
        *)
                echo  "cos innego"
                ;;
esac
```

1.5 Definiowanie funkcji

W bashu każda funkcja może przyjmować dowolną ilość parametrów pozycyjnych (w identyczny sposób obsługiwane są argumenty linii poleceń dla całego skryptu). Ilość parametrów znajduje się w zmiennej $#, lista wszystkich parametrów w $@, a do kolejnych parametrów możemy odwoływać się z użyciem $1, $2, itd.

```
f1() {
        echo "wywołano z $# parametrami, parametry to: $@"

        [ $# -lt 2 ] && return;

        echo -e "drugi: $2\npierwszy: $1"

        # albo kolejnych w pętli
        for a in "$@"; do  echo $a;  done

        # lub z użyciem polecenia shift
        for i in `seq 1 $#`; do
                echo $1
```

```
                shift # powoduje zapomnienie $1
                      # i przenumerowanie argumentów pozycyjnych o 1
                      # wpływa na wartości $@ $# itp
        done

        # funkcja może zwracać tylko wartość numeryczną -- tzw kod powrotu
        return 83
}
```

Zwróć uwagę że w nawiasach po nazwie funkcji nie podajemy przyjmowanych argumentów, natomiast puste nawiasy te są elementem składniowym i muszą wystąpić. Jeżeli zapisujesz definicję funkcji w jednej linii, np. `abc() { echo "abc"; }` to pamiętaj, że spacja po otwierającym nawiasie klamrowym jest obowiązkowa, podobnie jak średniki występujące po każdej (także ostatniej) instrukcji w ciele funkcji.

Wywołanie funkcji nie różni się niczym od wywołania programów czy instrukcji wbudowanych (możemy używać przekierowań strumieni wejścia, wyjścia, czy też przechwycić wyjście do zmiennej). Powyższą funkcję możemy wywołać np. w następujący sposób: `f1 a "b c"   d`

1.6 Grupowanie poleceń

Funkcje są przykładem grupowania poleceń – funkcja stanowi nazwany blok kodu, czyli nazwaną grupę poleceń. Polecenia w bashu możemy też grupować bez definiowania funkcji. W tym celu możemy zastosować nawiasy klamrowe (tak jak w definicji funkcji) lub nawiasy okrągłe.

Stosując nawiasy klamrowe musimy pamiętać (tak samo jak było to w przypadku funkcji) o spacji po otwierającym nawiasie klamrowym i średniku (lub nowej linii) przed zamykającym. Instrukcje podane w nawiasach klamrowych będą wykonane w bieżącej powłoce basha, czyli mogą modyfikować zmienne.

Polecenia podane w nawiasach okrągłych będą wykonane w podpowłoce, czyli ustawione lub zmodyfikowane w nich zmienne nie będą widoczne po zakończeniu bloku. Nawiasy okrągłe nie wymagają spacji i ostatniego średnika.

```
a=0;
{ echo abc; a=1; }
echo $a
(echo abc; a=2)
echo $a
```

Grupowanie poleceń jest przydatne na przykład w celu grupowania ich przy używaniu operatorów łączenia poleceń w oparciu o kod powrotu (`&&` i `||`), a także w celu przekazania standardowego wyjścia wielu poleceń w ramach jednego strumienia.

```
a=0;
{ echo AbC; echo abc; echo XyZ; a=1; } | grep b
echo $a
```

Zauważ że w tym wypadku nawiasy klamrowe zachowały się jak nawiasy okrągłe – modyfikacja zmiennej a nie jest widoczna po zakończeniu bloku. Wynika to z użycia przekierowania strumienia podobnie jak w sytuacji omawianej przy pętli while.

1.7 Wbudowane przetwarzanie napisów w bash'u

Wbudowane przetwarzanie napisów w bashu opiera się na odwołaniach do zmiennych w postaci `${}`:

- `${zmienna:-"napis"}` zwróci napis gdy zmienna nie jest zdefiniowana lub jest pusta

- `${zmienna:="napis"}` zwróci napis oraz wykona podstawienie zmienna=”napis” gdy zmienna nie jest zdefiniowana lub jest pusta
- `${zmienna:+"napis"}` zwróci napis gdy zmienna jest zdefiniowana i nie pusta
- `${#str}` zwróci długość napisu w zmiennej str
- `${str:n}` zwróci pod-napis zmiennej str od n do końca
- `${str:n:m}` zwróci pod-napis zmiennej str od n o długości m
- `${str/"n1"/"n2"}` zwróci wartość str z zastąpionym pierwszym wystąpieniem n1 przez n2
- `${str//"n1"/"n2"}` zwróci wartość str z zastąpionymi wszystkimi wystąpieniami n1 przez n2
- `${str#"ab"}` zwróci wartość str z obciętym ”ab” z początku
- `${str%"fg"}` zwróci wartość str z obciętym ”fg” z końca
- `${!x}` zwróci wartość zmiennej, której nazwa jest w zmiennej x

W napisach do obcięcia możliwe jest stosowanie shellowych znaków uogólniających, czyli `*`, `?`, `[abc]`, itd operator # i % dopasowują minimalny napis do usunięcia, natomiast operatory ## i %% dopasowują maksymalny napis do usunięcia. Należy pamiętać że wiele z powyższych zapisów jest rozszerzeniami basha niedostepnymi w podstaowej składni sh.

Przykład:

```
a=""; b=""; c=""
echo ${a:-"aa"} ${b:="bb"} ${c:+"cc"}
echo $a $b $c

a="x"; b="y"; c="z"
echo ${a:-"aa"} ${b:="bb"} ${c:+"cc"}
echo $a $b $c

x=abcdefg
echo ${#x} ${x:2} ${x:0:3} ${x:0:$((${#x}-2))}
echo ${x#"abc"} ${x%"efg"}
echo ${x#"ac"}  ${x%"eg"}

x=abcd.e.fg
echo ${x#*.} ${x##*.} ${x%.*} ${x%%.*}

y="aa bb cc bb dd bb ee"
echo ${y/"bb"/"XX"}
echo ${y//"bb"/"XX"}
```

```
aa bb
bb
x y cc
x y z
7 cdefg abc abcde
defg abcd
abcdefg abcdefg
e.fg fg abcd.e abcd
aa XX cc bb dd bb ee
aa XX cc XX dd XX ee
```

1.7.1 Wyrażenia regularne

Możliwe jest także korzystanie z wyrażeń regularnych. Polecenie `expr match $x 'wr1\(wr2\)wr3'` zwróci na stdout (wypisze) część `$x` pasującą do wyrażenia regularnego `wr2`, wyrażenia regularne `wr1` i `wr2` pozwalają na określanie części napisu do odrzucenia. Alternatywną składnią jest `expr $x : 'wr1\(wr2\)wr3'`

Przykład użycia jednego i drugiego wariantu składni do podziału napisu na część przed i po znaku równości:

```
z="ab=cd"
expr match $z '^\([^=]*\)='
```

```
expr $z : '^[^=]*=\(.*\)$'
```

Możliwe jest też sprawdzanie dopasowań wyrażeń regularnych poprzez (zwróć uwagę na brak cytowania wyrażenia regularnego):

```
[[ "$z" =~ ^([^=]*)= ]] && echo "OK"
```

1.7.2 printf

Możliwe jest także zaawansowane formatowanie napisów, konwertowanie liczb na napisy, w tym wypisywanie w różnych systemach liczbowych przy pomocy `printf`[5]:

```
printf "0o%o %d 0x%x\n" 0xf 010 3
```

```
0o17 8 0x3
```

1.8 Napis jako kod bashowy

1.8.1 nazwa polecenia w zmiennej

Jeżeli mamy w zmiennej nazwę polecenia do wykonania to możemy je wykonać uruchamiając po prostu tą zmienną:

```
a="echo"
$a ABC
```

```
ABC
```

Możemy nawet uruchamiać polecenia które zapisane mamy w zmiennej wraz z opcjami i argumentami:

```
a="echo -e abc\ndef ..."
$a ABC
```

```
abc
def ... ABC
```

Metoda ta nie pozwala jednak na podstawianie wartości zmiennych podanych w napisie w momencie uruchamiania go, itp.

1.8.2 eval

Polecenie wbudowane `eval` pozwala na wykonanie przekazanych do niego argumentów jako polecenie shellowe. Zapewnia ono podstawienie występujących w tym napisie zmiennych, itd.

Polecenia tego możemy użyć np. do z wartości zmiennej, której nazwę mamy w innej zmiennej (zamiast `${!x}`, które nie jest dostępne w czystym sh):

```
A='tekst do wypisania, $SHELL, `ls -d`'
B="A"

C=$( eval "echo \$$B" )
echo $C
```

```
tekst do wypisania, $SHELL, `ls -d`
```

Jeżeli chcielibyśmy aby kod znajdujący się w zmiennej A także został przetworzony (podstawiona wartość zmiennej oraz output komendy ls), możemy użyć eval dwukrotnie:

5. Instrukcja `printf` ma składnię opartą na tej funkcji z C, interpretuje ona także liczby zmiennoprzecinkowe.

```
A='tekst do wypisania, $SHELL, `ls -d`'
B="A"

C=$( eval eval "echo \$$B" )
echo $C
```

```
tekst do wypisania, /bin/bash, .
```

Dzięki użyciu eval możemy też budować kod bashowy (np. fragmenty konstrukcji case) w oparciu o zawartość zmiennych:

```
x=a

LISTA_WYBORU="a) echo AA;; b) echo BB;;"
INNE="e|f"

eval "case $x in
        $LISTA_WYBORU
        c) echo "CCC";;
        $INNE) echo inne;;
esac"
```

```
AA
```

1.8.3 envsubst

Innym poleceniem umożliwiającym podstawienie wartości zmiennych występujących w jakimś napisie jest envsubst. W odróżnieniu od eval nie wykonuje on podanego kodu, a jedynie przetwarza napis podstawiając wartości zmiennych (zatem jest bezpieczniejsza przy przetwarzaniu danych niewiadomego pochodzenia). Jako że jest to zewnętrzny program używane zmienne muszą być dla niego dostępne, czyli muszą być wyeksportowane.

```
x=12
A='tekst, ${x}, $SHELL, $x, `ls -d`'

C=`export x; echo $A | envsubst`
echo $C
```

```
tekst, 12, /bin/bash, 12, `ls -d`
```

1.9 Przekierowania, procesy, ...

Poznaliśmy już podstawy przekierowywania standardowych strumieni pomiędzy procesami oraz z użyciem plików. Mówiliśmy też o pewnych ograniczeniach jakie wiąża się z takimi przekierowaniami i programowaniem w bashu. Należy jednak wspomnieć jeszcze o kilku istotnych aspektach takich przekierowań.

1.9.1 cat, tee i <<

Przekierowania takie są często używane do tworzenia zewnętrznych plików z poziomu kodu skryptu bashowego. Wykonywanie takiej operacji z użyciem wielu instrukcji echo niekoniecznie jest wygodne, dlatego często stosowane jest następujące podejście:

```
cat > plik << EOF
wieloliniowa
treść zapisywana
```

```
do wskazanego pliku
EOF
```

Widzimy wywołanie komendy cat, z którym mogliśmy się już spotkać. Polecenie to wywołane z argumentami będącymi nazwami plików wypisze te pliki na standardowe wyjście. Wywołane bez argumentów przeczyta swoje standardowe wejście i wypisze je na standardowe wyjście (co, dzięki użyciu pzrzekierowania strumieni, tylko pozornie może wydawać się bezsensowne – na przykład wywołanie `cat > plik` pozwala nam wprowadzić treść do wskazanego pliku, przypomnij sobie także przesyłanie strumienowe plików z użyciem ssh).

Wyjście cat przekierowane jest do pliku, jednak zwraca tutaj uwagę inny operator - dwa znaki mniejszości (<<). Powoduje on że tekst podawany w kolejnych liniach będzie kierowany na standardowe wejście komendy po której wystąpił, aż do momentu napotkania linii zawierającej jedynie słowo podane po nim (w tym wypadku klasyczny `EOF`, ale może to być dowolne inne słowo - np. `KONIEC`, lub nawet ciąg znaków ze spacjami ujęty w cudzysłowa).

Innym poleceniem przydatnym przy manipulacji strumieniami jest tee. Podobnie jak cat kopiuje on swoje standardowe wejście na standardowe wyjście. Natomiast jeżeli poda mu się ścieżkę do pliku będzie on zapisywał te dane także do wskazanego pliku.

1.9.2 standardowe wejście/wyjście w miejscu ścieżki do pliku

Wiele programów jeżeli w miejscu w którym oczekuje ściezki do pliku otrzyma myślnik zinterpretuje to jako użycie w tym miejscu standardowego wejścia / wyjścia. Nawet jeżeli program nie wspiera tej konwencji możemy użyć specjalnych urządzeń reprezentujących te strumienie: `/dev/stdin`, `/dev/stdout`, `/dev/stderr`. Na przykład:

```
echo "ABC" > /dev/stderr
```

spowoduje wypisanie komunikatu ABC na standardowym wyjściu błędu.

Bash pozwala także na podstawienie standardowego wyjścia różnych komend w miejsce kilku plików bez potrzeby jawnego tworzenia plików tymaczasowych. Na przykład:

```
diff <(cat /etc/passwd) <(cat /etc/passwd-)
```

które poleceniu diff jako jeden plik postawia standardowe wyjście pierwszego cat, a jako drugi plik drugiego cat. Oczywiście zaprezentowane zastosowanie tego z poleceniami `cat` jest bezsensowne (prościej podać ścieżki do plików), ale gdyby występował tam już np. jakiś grep mogłoby to być użyteczne. Jest to rozszerzenie bashowe nie występujące w czystym sh.

1.9.3 polecenia wykonywane na zakończenie

Jak wiemy działający program może zostać zakończony w sposób niespodziewany, np. na skutek działania sygnałów. Dotyczy to również basha wykonującego jakiś skrypt powłoki. Możemy nawet samodzielnie zażądać przerwania wykonywania skryptu przy pierwszym błędzie (pierwszym poleceniu które zwróci nieobsłużony niezerowy kod powrotu) wydając polecenie `set -e`. Niekiedy chcemy móc w takiej sytuacji wykonać jakieś działania. Możliwe jest to z użyciem instrukcji `trap`, która pozwala na zdefiniowanie procedury obsługi sygnałów (oczywiście tych które mamy prawo obsłużyć) i innych zdarzeń. Na przykład:

```
trap '{ echo "to koniec"; }' EXIT
```

spowoduje wypisanie ”to koniec” w momencie kończenia pracy powłoki. Umieszczenie takiej instrukcji w kodzie skryptu spowoduje wypisanie tego komunikatu niezależnie od przyczyny zakończenia (dojście do końca skryptu wywołanie polecenia exit, przerwanie Control C, czy też otrzymanie innego przechwytywalnego sygnału).

2 System operacyjny

System operacyjny jest programem uruchamianym po starcie komputera (co prawda nie jako pierwszym, ale zastępującym / usuwającym z pamięci uruchomione wcześniej, więc najstarszym z działających).

2.1 Proces startu

Po otrzymaniu sygnału resetu (także przy uruchamianiu systemu - "Power-on Reset") procesor po inicjalizacji rejestrów zaczyna wykonywanie kodu znajdującego się pod jakimś ustalonym adresem (typowo w wbudowanej lub zewnętrznej pamięci typu ROM lub Flash). W zależności od danej architektury / procesrora może to być m.in.: bezpośrednio kod programu użytkownika, wbudowany bootloader danego procesora umożliwiający dalsze ładowanie np. z karty SD, zewnętrzny niskopoziomowy bootloader (np. u-boot).

W przypadku architektur zgodnych z x86 jest to BIOS, który po zakończeniu procesu inicjalizacji sprzętu i testów rozruchowych ładuje do pamięci kod znajdujący się w pierwszym sektorze dysku twardego (sektorze rozruchowym rozpoczynającym się od adresu zerowego) i uruchamia go (przekazuje do niego kontrolę). Znajduje się tam kod (lub tylko początek kodu) programu rozruchowego, którego zadaniem jest załadowanie systemu operacyjnego. W przypadku współczesnych systemów linuxowych jest to zazwyczaj GRUB.

Start systemu rozpoczyna się od załadowania do pamięci obrazu jądra wraz z parametrami oraz (opcjonalnie) initrd i przekazania kontroli do jądra przez program rozruchowy (np. GRUB). W przypadku jądra linuxowego i korzystania z initrd obraz ten przekształcany jest na RAM-dysk w trybie zapisu-odczytu i montowany jako rootfs z którego uruchamiany jest `/sbin/init` (którego podstawowym zadaniem jest zamontowanie właściwego rootfs). Po jego zakończeniu (lub od razu gdy nie używamy initrd) uruchamiany jest program wskazany w opcji `init=` jądra (domyślnie typowo `/sbin/init`) z rootfs wskazanego w opcji `root=` jądra. W opcji `init=` można wskazać dowolny program lub skrypt (uruchomiony zostanie z prawami root'a).

2.2 Rola systemu operacyjnego

System operacyjny jest oprogramowaniem odpowiedzialnym za zarządzanie zasobami systemu komputerowego (sprzętem, ale nie tylko) oraz uruchomionymi na nim aplikacjami. Do najistotniejszych zadań systemu operacyjnego zalicza się podział czasu procesora i szeregowanie zadań oraz zarządzanie pamięcią - w szczególności obsługa pamięci wirtualnej, najczęściej z wykorzystaniem mechanizmu stronicowania.

Oprócz tego system zajmuje się także zarządzaniem plikami, wejściem/wyjściem (najczęściej jest ono realizowane w oparciu o przerwania (IRQ), ale znane są także modele programowego we/wy polegającego na aktywnym czekaniu), obsługą urządzeń (wejście/wyjście, sterowniki, dostęp), obsługą sieci (stos protokołów sieciowych), itd.

Współczesne systemy korzystają z co najmniej dwóch poziomów pracy - uprzywilejowanego poziomu "nadzorcy" w którym działa jądro systemu operacyjnego oraz trybu użytkownika. Operacje I/O muszą odbywać się w trybie uprzywilejowanym. Również pamięć posiada obszar chroniony, w którym umieszczany jest m.in. tablica wektorów przerwań (inaczej zmiana adresu w tym wektorze mogłaby doprowadzić do przejęcia systemu w trybie uprzywilejowanym).

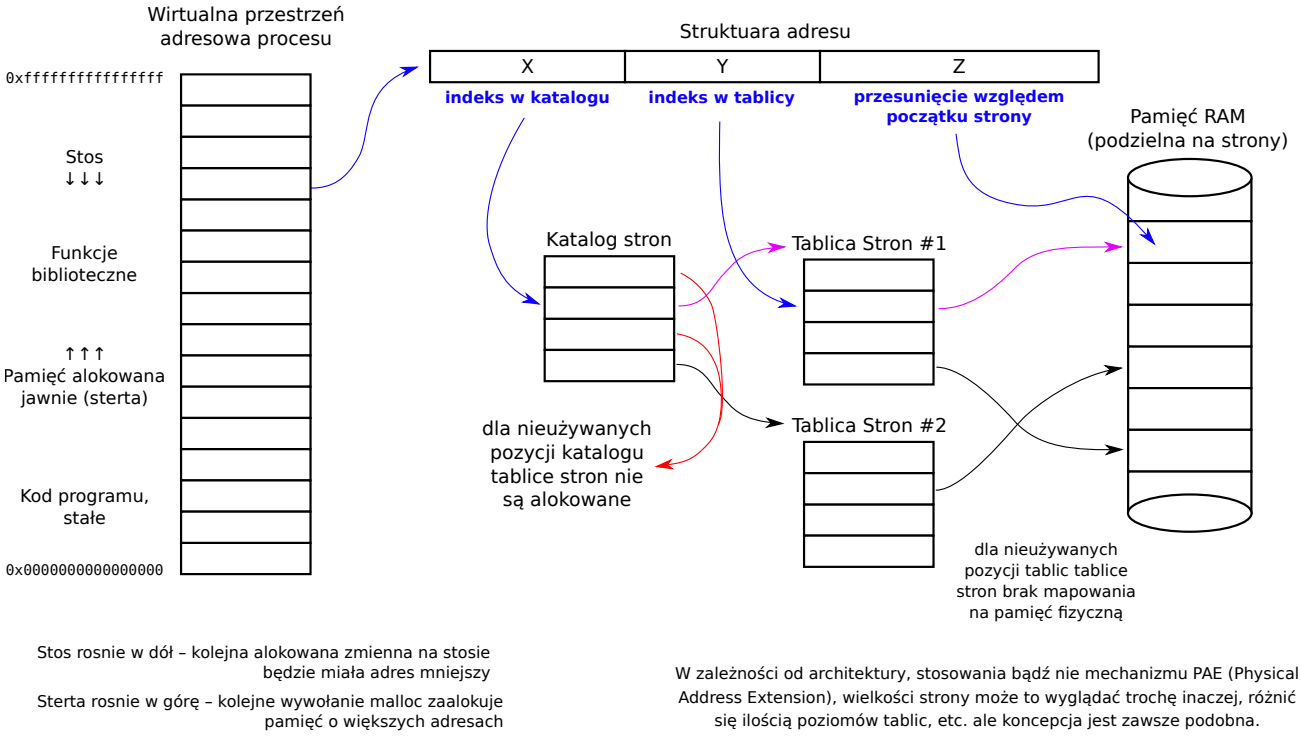
2.2.1 Zarządzanie procesami

Jednym z głównych zadań systemu operacyjnego jest zarządzanie procesami na nim uruchomionymi - obejmuje to zarówno czynności związane z ich tworzeniem oraz kończeniem, jak też szeregowanie zadań. To system operacyjny przejmuje okresowo procesor (z użyciem przerwania zegarowego), aby podjąć decyzję (w oparciu o jakiś algorytm szeregowania zadań) który/które z porcesów gotowych do działania (czyli takich, które aktualnie nie oczekują na "coś") ma dostać przydział czasu porcesora.

2.2.2 Zarządzanie pamięcią

Drugą podstawową funkcją systemu operacyjnego, wspomnianą na początku, jest zarządzanie pamięcią. Zarządzanie pamięcią polega na odpowiednim mapowaniu adresów logicznych (używanych przez procesy) na adresy fizyczne (używane przez procesor), korzysta ono z wsparcia sprzętowego ze strony procesora.

Jest to najczęściej realizowane w oparciu o wspomniany mechanizm stronicowania. Polega to na podziale pamięci dostępnej pamięci fizycznej na jednakowe bloki zwane ramkami oraz podziale pamięci logicznej na jednakowe bloki (o tej całej wielkości co ramki) zwane stronami. Strony które są wykorzystywane przez program są mapowane na dowolne ramki pamięci fizycznej (w przypadku gdy dana strona nie zamapowana - w zależności od okoliczności błąd braku strony lub błąd ochrony strony).



Rozwiązuje to problem fragmentacji zewnętrznej, polegającej na braku spójnego obszaru pamięci o żądanej długości pomimo iż łączna ilość wolnej pamięci jest dostateczna, jednak nie rozwiązuje problemu fragmentacji wewnętrznej, polegającej na przydzielaniu zbyt dużych fragmentów pamięci dla procesu (a wręcz można powiedzieć że go pogłębia).

Mechanizm ten wymaga trzymania tablicy wolnych ramek, tablicy stron dla każdego procesu (zawierającej przypisania mapowań stron danego procesu na ramki) oraz wykonywania tłumaczenia adresów logicznych (strona + przesunięcie na stronie). Także sama tablica stron procesu procesu może być stronicowania (mamy tablicę która informuje nas że przypisania stron w danym zakresie adresów są przechowywane w jakiejś ramce).

Strony i ramki mogą być współdzielone pomiędzy procesami (np. przy rozgałęzianiu procesu strony są kopiowane dopiero gdy zajdzie taka potrzeba). W przypadku braku miejsca w pamięci fizycznej wybrane strony nieaktywnego aktualnie procesu mogą być umieszczane na dysku (swap).

2.2.3 Literatura dodatkowa 😳

Było to bardzo krótkie i elementarne wprowadzenie w bardzo obszerny temat jakim są *systemy operacyjne*. Wszystkich zainteresowanym niskopoziomowym działaniem systemu operacyjnego zachęcamy do zapoznania się z kursem pisania OS[6] – niekoniecznie od razu pisania własnego systemu operacyjnego, ale przynajmniej zobaczeniem jak to się robi.

6. https://pl.wikibooks.org/wiki/Pisanie_OS

3 Wykład wideo

- *Wprowadzenie do programowania w bash'u* – https://www.youtube.com/watch?v=uJO42tK8EeE
- *Pętle, warunki i funkcje w bash'u* – https://www.youtube.com/watch?v=yf6e43ldGtg
- *Bash: obsługa napisów, eval i podsumowanie* – https://www.youtube.com/watch?v=GB7muE86q5E
- *Wprowadzenie do systemów operacyjnych* – https://www.youtube.com/watch?v=640nRcQjcCM

4 Literatura dodatkowa

- *Podręcznik Debiana* (https://www.debian.org/doc/manuals/debian-reference/)
- *Arch Linux Wiki* (https://wiki.archlinux.org/) – obszerny materiał przydatny także użytkownikom innych dystrybucji
- *Opis języka AWK* (https://archive.org/download/awkman-pl/awkman-pl.pdf)
- *Unix Toolbox* (http://cb.vu/unixtoolbox.xhtml) – zbiór przydatnych poleceń systemów unixowych
- *sed.sf.net - the sed $HOME* (http://sed.sourceforge.net/) – dużo informacji i przykładów dotyczących sed'a
- *Ściąga do VIM* (https://archive.org/download/virefcard-pl/virefcard-pl.pdf)
- *Ściąga do Debiana* (https://www.debian.org/doc/manuals/refcard/refcard.pl.pdf)
- *Ściąga do AWK* (https://archive.org/download/awkrefcard-pl/awkrefcard-pl.pdf)
- *Vademecum informatyki praktycznej* (http://vip.opcode.eu.org/) - zbiór materiałów na temat elektroniki i programowania, zawierający także dość rozbudowaną listę literatury dodatkowej.
- *Linux i Python w Elektronicznej Sieci* (https://ciekawi.icm.edu.pl/lpes) - strona domowa kursu *LPES*, zawierająca nagrania i skrypty do innych wykładów, skrypty ćwiczeniowe, itd.
- *OpCode.eu.org* (http://vip.opcode.eu.org/) - strona internetowa autora kursu *LPES*, zawierająca różne materiały z szeroko rozumianej inżynierii komputerowej i elektronicznej (część materiałów pokrywa się z zawartością skryptów z tego kursu, ale nie wszystkie)

5 Zadania

W rozwiązaniach zadań powinieneś korzystać z poleceń poznanych w ramach dzisiejszych zajęć oraz w ramach poprzednich dwóch zajęć z tematyki unixowej. Nie powinieneś stosować w tym celu Pythona, C, C++, itp. chyba że w ramach alternatywnych, nadprogramowych rozwiązań.

Zadanie 5.0.1

Napisz pętle, która wypisze wszystkie pliki nieukryte z bieżącego katalogu w postaci linków HTML, czyli: dla pliku o nazwie `ABC` powinna wypisać `<a href="ABC">ABC</a>`. Przedstaw zarówno rozwiązanie z użyciem pętli `for`, jak i pętli `while`.

Zadanie 5.0.2

Napisz warunek, który sprawdzi czy `/tmp/abc` istnieje i jest katalogiem.

Zadanie 5.0.3

Napisać funkcję przyjmującą dwa argumenty - liczbę i napis; funkcja ma wypisać napis tyle razy ile wynosi podana liczba.

Zadanie 5.0.4

Napisać funkcję przyjmującą jeden argument - liczbę kotów i wypisującą:

- "Ala ma kota" dla ilości kotów równej 1
- "Ala ma x koty" lub "Ala ma x kotów" gdzie dobrana jest poprawna forma, a pod x podstawiona podana w argumencie ilość kotów.

Dla uproszczenia należy założyć że podana ilość kotów jest w zakresie od 1 do 9.

Zadanie 5.0.5

Napisz polecenie które dla wszystkich plików z rozszerzeniem `.TXT` w bierzącym katalogu (bez podkatalogów) dokona zmiany ich nazwy zmieniając rozszerzenie na `.txt`, zachowując podstawową część nazwy bez modyfikacji. W rozwiązaniu nie korzystamy z polecenia `rename`.

Zadanie 5.0.6

Wyświetl z /etc/passwd linie w których UID (3 pole) ma warość >= 1000 nie korzystając z AWK. Jeżeli masz pomysł przedstaw więcej niż jedno rozwiązanie.

Zadanie 5.0.7

Napisz funkcję która przyjmuje dwa argumenty - napis wyszukiwany i napis go zastępujący oraz dokonuje rekurencyjnego wyszukania i zamiany tych napisów w wszystkich plikach w bieżącym katalogu.
Wskazówka 1: polecenie ***`sed`*** *z opcją* ***`-i`*** *i wskazaniem pliku modyfikuje zawartości tego pliku stosownie do poleceń wydanych sed'owi*
Wskazówka 2: dla uproszczenia możesz przyjąć że napisy te składają się jedynie z liter i cyfr.

6 Rozwiązania

Poniżej zamieszczone są przykładowe rozwiązania „głównych" zadań z tego skryptu wraz z komentarzami. Wiemy że zajrzenie do nich już przy pierwszej trudności jest kuszące, mimo to rekomendujemy przynajmniej podjąć uczciwą, co najmniej kilkunastominutową na każde z zadań, próbę rozwiązania tych zadania bez zaglądania do odpowiedzi.

Pamiętaj!: Samodzielne rozwiązanie problemu (wraz z wszystkimi trudnościami po drodze i popełnionymi błędami) jest dużo bardziej kształcące od nawet wielokrotnego przepisania gotowego rozwiązania, jednak nawet jednokrotne przepisanie rozwiązania jest bardziej kształcące od wielokrotnego przekopiowania go.

Zwróć uwagę że:

- rozwiązania te różnią się jedynie sposobem uzyskania listy plików ald której mają wypisać linki
- pętla for w każdym obiegu podstawia pod f kolejny element z listy nazw dopasowanych do gwiazdki (czyli wszystkich plików nieukrytych)
- pętla while listę plików dostaje na standardowe wejście (jeden plik na linię) i wczytuje każdą kolejną linię (czyli kolejną nazwę pliku) stosując komendę read – jest to bardzo standardowe rozwiązanie do przetwarzania standardowego wejścia linia po linii
- w obu wypadkach używamy takiego echo z napisem w podwójnych cudzysłowach (aby móc umieścić w nim zmienną), cudzysłowa które mają być wypisane zabezpieczamy odwrotnym

Rozwiązanie zadania 5.0.1

```
for f in *; do echo "<a href=\"$f\">$f</a>"; done
ls | while read f; do echo "<a href=\"$f\">$f</a>"; done
```

ukośnikiem
- wypisywanie można rozwiązać na kilka innych sposobów np.: `echo '<a href="'"$f"'">'"$f"'</a>'` – zadziała tak samo, ale wydaje się być to mniej czytelne

Rozwiązanie zadania 5.0.2

```
if [ -d /tmp/abc ]; then echo "jest katalogiem"; else echo "nie";
```

lub krócej:

```
[ -d /tmp/abc ] && echo "jest katalogiem";
```

Zwróć uwagę że:

- w celu warunkowego wypisania jakiejś informacji możemy użyć zarówno konstrukcji if, jak też łączenia poleceń, jednak w przypadku większej ilości poleceń objętych warunkiem konstrukcja z if jest bardziej czytelna
- sprawdzenie czy podana ścieżka istnieje i jest katalogiem odbywa się przy pomocy opcji -d, informacja ta celowo nie była podana w treści skryptu – należało to sprawdzić w dokumentacji systemowej (`man test`). **Czytanie dokumentacji jest ważne!**

Rozwiązanie zadania 5.0.3

```
f() { for i in `seq 1 $1`; do echo $2; done; }
```

Zwróć uwagę że:

- w nawiasach po nazwie funkcje nie piszemy nic na temat jej argumentów - one są puste
- do argumentów odwołujemy się poprzez dolar numer argumentu
- do n krotnego powtórzenia czynności używamy pętli for która iteruje po liście liczb zwracanej przez seq
- seq objęta jest znakami ` oznaczającymi że należy wykonać podany w nich kod i podstawić w to miejsce jego standardowe wyjście, nie należy ich mylić z apostrofami używanymi do napisów (')
- spacja po oraz średnik (lub nowa linia) przed są istotne składniowo

```
koty() {
        case $1 in
                1) echo "Ala ma kota";;
                2|3|4) echo "Ala ma $1 koty";;
                *) echo "Ala ma $1 kotów";;
        esac
}
```

Rozwiązanie zadania 5.0.4

```
for f in *.TXT; do mv "$f" "${f%.TXT}.txt"; done

# albo:

for i in *.TXT; do mv "$i" "$(basename $i .TXT).txt"; done
```

Do iteracji po plikach używamy pętli for, możemy jej od razu podać wyrażenie określające po jakich plikach ma przebiegać. Nie potrzebujemy tutaj konstrukcji typu `for f in `ls *.TXT`, aczkolwiek w bardziej złożonych przypadkach może ona być użyteczna. Na przykład gdy chcemy pominąć pliki

w nazwie których wystąpuje XYZ: `for f in `ls *.TXT | grep -v XYZ`` (pod warunkiem że nie mamy plików ze spacjami w nazwie - wtedy lepiej zrobić `for f in *.TXT` i filtrować dalną nazwę w każdym obiegu pętli lub użyć potoku `ls | grep | while read ...`).
W takim przypadku `${f%.TXT}.txt` jest lepsze od `${f/.TXT/.txt}` lub `${f//.TXT/.txt}` gdyż usuwa tylko ciąg `.TXT` wystepujący na końcu napisu. `aa.TXT.xyz.TXT` jest poprawną nazwą pliku i zgodnie z warunkami zadania powinna być zmieniania na `aa.TXT.xyz.txt`, a nie na `aa.txt.xyz.TXT` lub `aa.txt.xyz.txt`.

Rozwiązanie zadania 5.0.6

```
while read l; do x=$(echo $l| cut -f3 -d:);
    [ $x -ge 1000 ] && echo $l;
done < /etc/passwd
```

```
IFS=":"; while read p1 p2 p3 pX; do
    if [ $p3 -ge 1000 ]; then echo "$p1:$p2:$p3:$pX"; fi
done < /etc/passwd; unset IFS
```

```
egrep '^([^:]*:){2}[0-9]{4}' /etc/passwd
```

Warto zwrócić uwagę że rozwiązanie z awk (`awk -F: '$3>=1000 {print $0}' /etc/passwd`) jest znacznie prostsze.

Rozwiązanie zadania 5.0.7

```
replace() {
    grep -lR "$1" . | while read f; do
        sed -e "s@$1@$2@g" -i $f;
    done;
}
```

Zamiast `grep -lR "$1" .` można by napisać `grep -R "$1" . | cut -f 1 -d: | uniq`, w większości przypadków zadziała tak samo, ale jest bardziej skomplikowane, wymaga więcej zasobów i będzie miało problem z nazwami plików zawierającymi dwukropek.