# Notatki 2

Opowieści, skrypty

**Edytor tekstu Vim** – warto znać chociaż jeden edytor tekstu nie wymagający środowiska graficznego. Bardzo ułatwia to pracę na zdalnych komputerach. Do takich edytorów należą chociażby `vi`, `vim`, `emacs` i `nano`. Na wykładzie będę używał intensywnie edytora `vim`.

**Ściągawka do Vim-a:**

- `Esc` – tryb poleceń
- `i/o` – tryb edycji (`o` zaczyna w nowej linii)
- `Ins` – tryb edycji (nadpisywanie/wpisywanie)
- `v/V/Ctrl+v` – tryb „wizualny” (zwykły/liniowy/blokowy), przydaje się do zaznaczania i wycinania
- `:w <plik>` – zapisz do pliku (bez podania pliku zapisuje w bieżącym)
- `:q` – zamknij plik
- `:wq` – zapisz i zamknij
- `:r <plik>` – wklej zawartość pliku
- `!` – wymuś (np. :q! wychodzi z programu bez zapisu)
- `y` – skopiuj (`yy` – bieżącą linię)
- `p/P` – wklej za/przed kursorem
- `d` – usuń (`dd` – bieżącą linię)
- `:<numer>` – skocz do linii o podanym numerze
- `gg` – skocz na początek pliku
- `G` – skocz na koniec pliku
- `/<coś>` – wyszukaj coś
- `?<coś>` – wyszukaj coś, ale szukając wstecz
- `:s/a/b/` – zamień pierwsze wystąpienie „a” na „b” w bieżącej linii
- `:s/a/b/g` – jak wyżej, ale zamień wszystkie wystąpienia „a”
- `:%s/a/b/g` – jak wyżej, ale w całym pliku
- `:'<,'>s/a/b/g` – jak wyżej, ale w zaznaczonych (w trybie wizualnym) liniach
- `u/ctrl+r` – cofnij/anuluj cofnięcie

**Podstawowe operacje przeszukiwania w Linuxie**

Konwencjonalnie symbolem zachęty (ang. *prompt*) „`$`” oznacza się polecenia powłoki wykonywane przez zwykłego użytkownika (roota oznacza się „`#`”). Nie wpisuje się tego symbolu w powłoce, ma on w literaturze jedynie znaczenie informacyjne. Większość współczesnych systemów ma znacznie bardziej rozbudowany symbol zachęty (np. `użytkownik@komputer:bieżący_katalog$`). Można go często zmodyfikować w pliku `.bashrc`. Znak „`#`” w Bashu oznacza komentarz, z tym jednym wyjątkiem, gdy następuje po nim wykrzyknik i ma to miejsce na samym początku pliku (znak zachęty omówiony przed chwilą nie stanowi elementu składni Basha, jest jedynie ozdobą). Wówczas to tzw. *shebang* i informuje system za pomocą jakiej powłoki powinien zostać wykonany (jeżeli nie robimy tego jawnie).

```
$ cat /proc/cpuinfo                          #wypisz zawartość pliku /proc/cpuinfo

$ grep processor /proc/cpuinfo               #wyszukaj linie zawierające wyrażenie
                                             #"processor" w pliku /etc/passwd

$ grep -c processor /proc/cpuinfo            #to samo, ale podaj jedynie liczbę trafień

$ grep -B 3 -A 1 "model name" /proc/cpuinfo  #tym razem szukamy "model name" i chcemy
                                             #też 3 linie przed i 1 po

$ grep -r Linux Technologie/                 #wyszukaj linie zawierające słowo "Linux"
                                             #we wszystkich plikach w katalogu "Technologie"

$ grep -c Linux *.tex                        #policz linie ze słowem Linux we wszystkich
                                             #plikach (w bieżącym katalogu) o nazwie kończącej
                                             #się ciągiem znaków ".tex"

$ find Technologie/ -name "n*.tex"

$ find Technologie/ -type d -name "w*"
# rm -fr --no-preserve-root /                #NIE UŻYWAĆ! podałem to tylko by zademonstrować
                                             #znak "#" jako prompt roota, a także ku przestrodze;
                                             #komenda ta usunie wszystko z dysku... a w rzadkim,
                                             #najgorszym przypadku zepsuje płytę główną - w
                                             #niektórych systemach zmienne oprogramowania płyty
```

```
                                        #głównej (efivars) były montowane w trybie
                                        #umożliwiającym zapis i kasowanie
```

**Skrypty powłoki**

Polecenia, które wpisujemy w powłoce, bywają powtarzalne i łatwe do zautomatyzowania. Umożliwiają to skrypty. Wystarczy zapisać polecenia w pliku i uruchomić go za pomocą programu powłoki. Np. zapiszmy w pliku `skrypt0.sh` co następuje:

```
#!/bin/bash
echo "To jest mój pierwszy skrypt!"
pwd
date
```

Po czym wywołajmy go przez

```
$ bash skrypt0.sh
```

albo

```
$ chmod u+x skrypt0.sh                          #nadaj uprawnienia wykonywania właścicielowi pliku
$ ./skrypt0.sh
```

Skrypt ten wypisuje na ekran żądany tekst za pomocą polecenia `echo`, następnie podaje w jakim katalogu został wywołany oraz datę.

Możemy stosować zmienne, np.

```
#!/bin/bash
i=0
echo "Zmienna i zawiera wartość: $i"
i=$((i+1))
echo "A teraz i zawiera wartość: $i"
echo "A teraz i zawiera wartość: $((3*i))"
echo "A teraz i zawiera wartość: $(bc <<<"scale=6;i/3")
echo "A teraz i zawiera wartość pi: $(bc -l <<<"4*a(1)")
i=$(date)
echo "A teraz i zawiera wartość: $i"
```

Warto zwrócić uwagę, że możemy łatwo wykonywać działania na liczbach całkowitych – dolar i podwójne nawiasy: `$((wyrażenie))`, ale nie na liczbach rzeczywistych. By osiągnąć to ostatnie, posługujemy się programem `bc` (*basic calculator*, bardzo przydatny niezależnie od skryptów). W Bashu możemy wstawić wynik jakiegoś polecenia w dowolne miejsce, zawierając jego wywołanie w pojedynczych nawiasach z dolarem: `$(polecenie)`. Domyślnie `bc` zwraca wynik z liczbą miejsc po przecinku (którą oznacza jako `scale`) dziedziczoną z danych wejściowych. Są jednak wyjątki, takie jak np. dzielenie, które stosują się do globalnej wartości tego parametru (`scale=0`). Dlatego też oprócz działania podaliśmy jawnie tę wartość. Jeżeli chcemy wywołać jakieś bardziej skomplikowane funkcje, musimy dodać opcję `-l`, która wczytuje bibliotekę matematyczną (i dodatkowo ustawia `scale=20`). W powyższym przykładzie wyznaczamy wartość liczby $\pi = 4\arctan(1)$. Potrójny znak `<<<` oznacza tzw. *here-string*, poprzez który możemy przekazać programowi jakiś ciąg znaków (więcej o tym za chwilę).

W tym punkcie należy zauważyć jeszcze jedną osobliwość składni Basha. Mianowicie jeśli użyjemy w powyższym skrypcie pojedynczych cudzysłowów (`'`) zamiast podwójnych, znak dolara przestanie być traktowany jako znak specjalny i np. `echo 'A teraz i zawiera wartość: $i'` zamiast wypisania wartości zmiennej `i` po prostu wypisze na ekran znaki `$i`.

Jak przystało na porządny język programowania/skryptowania, mamy do dyspozycji pętle for

```
#!/bin/bash
for i in a b c d e
do
  echo "$i hello"
done
```

By iterować po automatycznie stworzonym zakresie liczb, w literaturze często spotyka się rozwiązanie wykorzystujące zewnętrzny program `seq`:

```
#!/bin/bash
for i in $(seq 1 5)
do
```

```
  echo "$i hello"
done
```

Nie ma jednak potrzeby, by to robić – Bash ma wbudowane zakresy liczb:

```
#!/bin/bash
for i in {1..5}
do
  echo "$i hello"
done
```

Możemy też skakać co kilka:

```
#!/bin/bash
for i in {1..5..2}
do
  echo "$i hello"
done
```

Dostępne są także pętle w stylu języka `C`:

```
for ((i=1;i<=5;i++))
do
  echo "${i}hello"
done
```

Warto tu nadmienić, że jeśli zmienna występuje w niejasnym kontekście (np. jest sklejona z jakimś dalszym tekstem, więc Bash nie wiedziałby gdzie się kończy nazwa zmiennej, a zaczyna co innego), można to doprecyzować przez wzięcie jej w nawiasy klamrowe, jak powyżej.

Oczywiście istnieje też konstrukcja warunkowa `if`:

```
#!/bin/bash
for i in {1..5}
do
  if [ $i == 3 ]
  then
    echo "Pozdrawiam z $i"
  elif [ $i == 4 ]
  then
    echo "Nic tu nie powiem"
  else
    echo "$i.hello"
  fi
done
```

**Strumienie i potoki w powłoce**

Bardzo wygodnym aspektem pracy w powłoce jest możliwość łączenia ze sobą programów przez używanie danych wyjściowych jednego jako wejściowe drugiego. Kanały przesyłania danych określa się tu mianem *strumieni*. Do pracy ze strumieniami możemy wykorzystać symbol „|" – potok (ang *pipe* (jak widać metafory hydrauliczne będą tutaj dość powszechne).

```
$ top -bn 1 | grep bash                    #wypisz procesy poleceniem top i znajdz linie z "bash"

$ grep -cr Linux wykład/ | grep -v ':0'    #wypisz liczbę linii ze słowem "Linux" dla każdego
                                           #pliku w katalogu "wykład", a następnie wśród nich
                                           #wyszukaj te, gdzie podano niezerową liczbę linii

$ grep -o sse /proc/cpuinfo | wc -l        #wyszukaj wszystkie wystąpienia wyrazu "sse" w
                                           #pliku "/proc/cpuinfo", a następnie policz je

$ echo "2 + 2" | bc                        #wypisz wyrażenie "2 + 2" - ale zamiast na ekran
                                           #przekaż je do obróbki kalkulatorowi bc
```

Jak widzieliśmy poprzednio, ostatni przykład możemy zrealizować także przez *here-string*:

```
$ bc <<< "2 + 2"
```

Powłoka pozwala nam też na przekierowanie wyniku polecenia do jakiegoś pliku. Służy do tego znak „>”.

```
$ top -bn 1 > top.txt                          #wypisz procesy i zapisz do pliku

$ top -b > top.txt                             #wypisuj procesy cyklicznie i zapisuj je do pliku
$ tail top.txt                                 #wypisz koniec pliku top.txt
$ tail -f top.txt                              #wypisuj na bieżąco zmieniający się plik top.txt
$ top -b | tee top.txt                         #wypisuj procesy cyklicznie - jednocześnie na ekran
                                               #i do pliku top.txt
```

Użycie „>” spowoduje, że w momencie wywołania zawierającego je polecenia plik, do którego będziemy zapisywać zostanie wyczyszczony, jeśli wcześniej istniał. Jeżeli chcielibyśmy jedynie dopisywać zawartość do istniejącego pliku, zachowując to co wcześniej zawierał, to powinniśmy zamiast „>” użyć „>>”.

Utwórzmy teraz `skrypt1.sh`, zawierający co następuje:

```
#!/bin/bash
echo "Przeczytam teraz plik, którego nie ma"
cat /home/cthulhu                               #wybierzmy jakiś plik, o którym wiemy, że nie
                                                #istnieje... a przynajmniej taką mamy nadzieję
```

Tu mała uwaga – rozszerzenie „`.sh`” dodaliśmy jedynie by ułatwić użytkownikowi (i systemowi) identyfikację pliku. Możemy je zmienić na cokolwiek innego lub pominąć. Jedynym mankamentem będzie ewentualne wprowadzenie użytkownika w błąd – a także systemu, gdy spróbujemy otworzyć taki skrypt w okienkowej przeglądarce (lub równoważnie poleceniem `xdg-open`).

```
$ mv skrypt1.sh skrypt1.pdf
$ file skrypt1.pdf                             #podaj informacje o pliku
$ bash skrypt1.pdf                             #wykonaj skrypt, nawet z rozszerzeniem .pdf
$ xdg-open skrypt1.pdf                         #otwórz programem domyślnym
```

Wróćmy jednak do samego skryptu:

```
$ chmod u+x skrypt1.sh
$ ./skrypt1.sh
$ ./skrypt1.sh > out                           #zapiszmy wynik do pliku "out"
```

Okaże się, że nadal część danych trafia nie do pliku, a na ekran. Wynika to z tego, że system oddzielnie traktuje komunikaty o błędach. Kierowane są one do strumienia `stderr`, który oznaczany jest deskryptorem „2” („zwykłe” wyjście, `stdout`, oznaczane jest „1”, a wejście, `stdin`, to „0”). Możemy przekierować `stderr` do oddzielnego pliku, albo nawet i tego samego:

```
$ ./skrypt1.sh > out 2>err                     #zapiszmy błędy do pliku "err"
$ ./skrypt1.sh > out 2>&1                      #dołączmy błędy do standardowego wyjścia
```

**Uruchamianie skryptów w tle**

Skrypty możemy również uruchamiać w tle, by nie zajmowały nam niepotrzebnie terminala. Służy do tego znak „`&`” na końcu wiersza. Zmodyfikujmy trochę plik ze skryptem (nazwijmy go też `skrypt2.sh`).

```
#!/bin/bash
while true
do
 echo "Przeczytam teraz plik, którego nie ma"
 cat /home/cthulhu
 sleep 0.5
done
```

Warto tu zwrócić uwagę na użycie konstrukcji `while`, która jest swego rodzaju połączeniem pętli `for` i instrukcji warunkowej `if`. Jako warunek do sprawdzenia dałem `true`, które z definicji jest zawsze prawdziwe, więc pętla będzie chodziła w nieskończoność. Polecenie `sleep 0.5` powoduje, że program czeka przez pół sekundy po każdej iteracji.

```
$ ./skrypt2.sh > out 2>&1 &
```

Możemy podejrzeć jak poprzednio plik `out` (`tail -f`) – zobaczymy, ze pojawiają się w nim nowe wpisy. Aby zobaczyć zadania, które uruchomiliśmy w tle, możemy użyć polecenia:

```
$ jobs -l
```

Zwróci nam ono coś podobnego do:

```
[1]+ 651530 Running                 ./skrypt2.sh > out 2>&1 &
```

Liczba `651530` to numer procesu. Wywołując `kill 651530` możemy go zakończyć.

**Pierwsze spotkanie z narzędziami sed i awk**

W tym miejscu warto wspomnieć o dwóch potężnych narzędziach, których będziemy więcej używać na przyszłych wykładach – `sed` oraz `awk`. Pozwalają one na manipulowanie danymi. Ich funkcjonalność w znacznej mierze się pokrywa, ale zazwyczaj `sed` jest bardzo wygodny do pracy na wierszach (np. podmiany wyrażeń), zaś `awk` – na kolumnach (np. wypisywanie tylko wybranych kolumn z pliku).

```
$ top -bn 1 | awk 'NR>6 && NR<18 {print $12,$9}'
```

```
$ sed 's/o/x/g'<<<"programowanie"
```

W przypadku polecenia `awk` przefiltrowaliśmy dane wyjściowe polecenia `top` tak, aby najpierw pokazana była dwunasta kolumna (z nazwą procesu), a następnie dziewiąta (z wykorzystaniem CPU przez dany proces). Aby pozbyć się nagłówka (zajmującego pierwsze 6 linii) oraz wypisać tylko kilka najistotniejszych procesów, posłużyliśmy się zmienną `NR` zawierającą liczbę przetworzonych przez `awk` rekordów (wierszy). Chcemy, by program zwracał cokolwiek jedynie dla numeru wiersza większego niż `6` i mniejszego niż `18`.

W przypadku `sed` dokonaliśmy zamiany (`s`) liter „o” na „x” w słowie „programowanie”. Litera `g` (*global*) powoduje zmianę wszystkich wystąpień, a nie tylko pierwszego (co uzyskalibyśmy pomijając ją). Jak widać, jest to składnia prawie taka sama jak w edytorze `vim`. Oba programy mogą oczywiście działać nie tylko na strumieniach (jak tutaj), ale także na plikach. To i bardziej zaawansowane przykłady omówimy później – na razie warto pamiętać, że takie narzędzia istnieją.