Kurs administrowania systemem Linux

Zajęcia nr 3: Powłoka systemowa

Instytut Informatyki Uniwersytetu Wrocławskiego

13 marca 2025

Podstawy korzystania z powłoki systemowej

Idiomatyzmy basha

- *Sourcing*
- Środowisko a podprocesy: używanie zmiennych w podprocesach
- Problemy podziału na słowa: spacje w nazwach plików
- Usuwanie pustych słów
- Błędy w trakcie wykonania skryptów
- Globy
- Przekierowania

Wywoływanie programów

- Opcje programów i `getopt`
- Dokumentacja: `man` i `info`

source

- `source` *`script`* — wykonanie skryptu *`script`* w *bieżącej* powłoce.
- Jedyny sposób, żeby zmienić środowisko bieżącego procesu.
- Skrypty startowe `/etc/bash.bashrc`, `/etc/profile`, `~/.bashrc` itp. są `source`'owane.
- Biblioteki funkcji wykorzystywanych przez wiele skryptów,
  zob. np. `/lib/lsb/init-functions`.
- Skrót: `.` (kropka) — nie używać w skryptach (podobnie jak np. `` `...` `` zamiast `$(...)`).
- Idiom: `. ~/.bashrc`
- Uwaga na zaśmiecanie środowiska! Zmienne lokalne dla skryptu powinny zostać `unset`.
  Uwaga na kolizje zmiennych lokalnych skryptu ze zmiennymi globalnymi.

Pliki source'owane a wykonywane (w podprocesie)

Plik source'owany (np. .profile)

```
...
TMP="zmienna lokalna"
ORIG_IFS=$IFS
IFS=$' \t'
...
IFS=$ORIG_IFS
unset ORIG_IFS TMP
```

Uwaga na zmienne istniejące już w środowisku (np. TMP)!

Plik wykonywany w podprocesie (skrypt)

Modyfikacje zmiennych środowiskowych są ograniczone tylko do wykonania tego skryptu.

Pliki konfiguracyjne skryptów

Skrypt

```
...
MYCONFIG=~/.myconfig
VAR1="script default for VAR1"
VAR2="script default for VAR2"
[ -f $MYCONFIG ] && source $MYCONFIG
...
```

Plik ~/.myconfig

```
# Opis zmiennej VAR1 i zakomentowana wartość domyślna
# VAR1="script default for VAR1"
# Opis zmiennej VAR2 i zakomentowana wartość domyślna
# VAR2="script default for VAR2"
```

Ten sam trik można stosować dla programów:

```
[ -x /usr/bin/mandb ] && /usr/bin/mandb
```

Nie róbcie tego w domu

```
#!/bin/bash

if [ -z "$N" ]
then
    N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | . $0 $1
fi
```

Nie róbcie tego w domu

```
#!/bin/bash

if [ -z "$N" ]
then
    N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | source $0 $1
fi
```

Nie róbcie tego w domu

```
#!/bin/bash

if [ -z "$N" ]
then
    N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | source $0
fi
```

Nie róbcie tego w domu

```
#!/bin/bash

if [ -z "$N" ]
then
    N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | $0 $1
fi
```

Nie róbcie tego w domu

```
#!/bin/bash

if [ -z "$N" ]
then
   export N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | $0 $1
fi
```

Nie róbcie tego w domu

```
#!/bin/nobash

if [ -z "$N" ]
then
   export N=$1
else
   cat
fi

if [ $N -gt 0 ]
then
   N=$((N-1))
   echo "$BASHPID $0 $1 $N" | $0 $1
fi
```

Skrypt rekurencyjny a funkcja rekurencyjna

```
#!/bin/bash

N=$1

function recursive {
   cat
   if [ $N -gt 0 ]
   then
      N=$((N-1))
      echo "$BASHPID $0 $1" | recursive
   fi
}

echo "Start: $BASHPID $0 $1" | recursive
```

Przypisania zmiennych

```
$ DZIEWCZYNKA=Ala
$ INWENTARZ="$DZIEWCZYNKA ma kota"
$ echo $INWENTARZ
```

Przypisania zmiennych

```
$ DZIEWCZYNKA=Ala
$ INWENTARZ="$DZIEWCZYNKA ma kota"
$ echo $INWENTARZ
Ala ma kota
$ DZIEWCZYNKA=Ola
$ echo $INWENTARZ
```

Przypisania zmiennych

```
$ DZIEWCZYNKA=Ala
$ INWENTARZ="$DZIEWCZYNKA ma kota"
$ echo $INWENTARZ
Ala ma kota
$ DZIEWCZYNKA=Ola
$ echo $INWENTARZ
Ala ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:

  ```
  $ MSG=Run ls; echo $MSG
  ```

Przypisania zmiennych

```
$ DZIEWCZYNKA=Ala
$ INWENTARZ="$DZIEWCZYNKA ma kota"
$ echo $INWENTARZ
Ala ma kota
$ DZIEWCZYNKA=Ola
$ echo $INWENTARZ
Ala ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:

  ```
  $ MSG="Run ls"; echo $MSG
  ```

Przypisania zmiennych

```
$ DZIEWCZYNKA=Ala
$ INWENTARZ="$DZIEWCZYNKA ma kota"
$ echo $INWENTARZ
Ala ma kota
$ DZIEWCZYNKA=Ola
$ echo $INWENTARZ
Ala ma kota
```

- Przed przypisaniem zmiennej są wykonywane *niektóre* rozwinięcia, w tym rozwinięcia zmiennych.
- Po rozwinięciach podział na słowa nie jest wykonywany, ale uwaga na spacje w oryginalnej instrukcji:

  ```
  $ MSG="Run ls"; echo $MSG
  ```

- Dołączanie na koniec: `+=`

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ((FAC *= N--))
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
???
```

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ((FAC *= N--))
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
Factorial of 5 equals 120
```

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ((FAC *= N--))
done | cat
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
???
```

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ((FAC *= N--))
done | cat
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
Factorial of 5 equals 1
```

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ( ((FAC *= N--)) )
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
???
```

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ( ((FAC *= N--)) )
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
```

zapętlenie

Zmienne a instrukcje wykonywane w podprocesach

fac.sh

```
N=$1
FAC=1
while ((N>1))
do
   ((FAC *= N--))
done
echo "Factorial of $1 equals $FAC"
```

```
$ bash fac.sh 5
Factorial of 5 equals 120
```

Zasięg zmiennych w bashu nie jest zasięgiem leksykalnym!

Względne i bezwzględne ścieżki do poleceń w skryptach

- Polecenie *cmd* w skrypcie zostanie skojarzone z tym, co aktualnie jest związane z tą nazwą (np. zostanie rozwinięty alias *cmd*, a następnie zostanie wykonane podstawione polecenie, np. polecenie wbudowane, funkcja lub program).
- Polecenie `/usr/bin/`*cmd* w skrypcie wykona zawsze program `/usr/bin/`*cmd*.
- Polecenie `command` *cmd* wykona polecenie wbudowane lub program (zgodnie z bieżącym ustawieniem zmiennej `PATH`). Alias nie zostanie wcześniej rozwinięty (bo polecenie *cmd* nie jest pierwszym tokenem w wierszu), a funkcje zostaną pominięte.
- Polecenie `builtin` *cmd* wykona polecenie wbudowane *cmd*.
- Problem właściwej ścieżki do programu.
- To samo dotyczy `#!`. Tu można zrobić tak:

  ```
  #!/usr/bin/env bash -
  ```

- Polecenie `type -a` *cmd* wypisze wszystkie możliwe skojarzenia nazwy *cmd*. Pierwsze znalezione to to, które będzie wykonane, jeśli wpiszemy takie polecenie.

Parametryzowanie skryptu

Konfiguracja na początku skryptu

```
RSYNC=/usr/bin/rsync
...
if [ -n "$DRY_RUN" ]
then
   CMD="echo rsync"
else
   CMD=$RSYNC
fi
$CMD -av --delete $FROM $TO
```

Dobre również podczas pracy interaktywnej:

Sprawdź *zanim* wykonasz

```
$ for i in *.md; do echo mv $i $(basename $i .md).txt; done
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
for FILE in ./*.txt
do
   echo "$FILE"
done
```

```
$ bash text.sh
???
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
for FILE in ./*.txt
do
   echo "$FILE"
done
```

```
$ bash text.sh
./spacje w nazwie pliku.txt
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
for FILE in $(find . -name '*.txt' -print)
do
   echo "$FILE"
done
```

```
$ bash text.sh
???
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
for FILE in $(find . -name '*.txt' -print)
do
   echo "$FILE"
done
```

```
$ bash text.sh
./spacje
w
nazwie
pliku.txt
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
IFS=$'\n'
for FILE in $(find . -name '*.txt' -print)
do
   echo "$FILE"
done
```

```
$ bash text.sh
./spacje w nazwie pliku.txt
```

Problemy podziału na słowa

```
$ touch 'spacje w nazwie pliku.txt'
```

text.sh

```
IFS=$'\n'
for FILE in $(find . -name '*.txt' -print)
do
   echo "$FILE"
done
```

```
$ bash text.sh
./spacje w nazwie pliku.txt
```

Podział na słowa zależy od kontekstu obliczeń i zmiennej IFS.

Rozwinięcia (przypomnienie)

- rozwinięcia nawiasów wąsatych, np. `file{1,2,3}`, `file{1..10}`,
- rozwinięcia tyldy, np. `~/Downloads/`,
- rozwinięcia zmiennych, np. `$HOME`,
- podstawienia instrukcji, np. `$(cat file.txt)`,
- podstawienia procesów, np. `<(pdftops file.pdf -)`,
- rozwinięcia arytmetyczne, np. `$((N+1))`,
- (powtórny) podział na słowa,
- rozwinięcia nazw plików (*globów*), np. `file?-*.txt`.

Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
```

Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
1 2 3 4 5
$ x=5; echo {1..$x}
```

Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
1 2 3 4 5
$ x=5; echo {1..$x}
{1..5}
```

A jak spowodować rozwinięcie `$x`?

Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
1 2 3 4 5
$ x=5; echo {1..$x}
{1..5}
```

A jak spowodować rozwinięcie `$x`?

```
$ x=5; eval echo {1..$x}
```

Kolejność wykonywania rozwinięć

- Sporo zależności, np. podstawienia instrukcji powinny być wykonywane *przed* rozwinięciami globów, bo np.

```
echo $(cd ..; ls -1 *)
```

- Problem:

```
$ echo {1..5}
1 2 3 4 5
$ x=5; echo {1..$x}
{1..5}
```

A jak spowodować rozwinięcie `$x`?

```
$ x=5; eval echo {1..$x}
1 2 3 4 5
```

- W powyższym przykładzie widać, czemu przyjęto, że niepoprawne rozwinięcia nawiasów wąsatych, np. `{1..$x}`, pozostają niezmienione — mogą stać się poprawne później, po kolejnych rozwinięciach.

Uwagi o problemie podziału na słowa

- Znaki oddzielające słowa opisuje zmienna `IFS` (*Internal Field Separator*).
  Domyślnie `$' \t\n'`.
- Podział na słowa następuje *po* wykonaniu rozwinięć *z wyjątkiem rozwinięć nazw plików*.
- Opcje `-print0` programu `find` i `-0` programów `xargs` i `parallel`.
- Znak `'\0'` nie działa w `$IFS` — jest terminatorem napisu.
- Opcja `-exec` programu `find` często pozwala wykonać całą pracę podczas wywołania `find`.
- W systemach plików `ext234` nazwa pliku jest ciągiem co najwyżej 255 bajtów różnych od `0x00` i `0x2f` (`'/'`).
- Znak `'\n'` jest dopuszczalny w nazwach plików!
- Nazwy plików w kodowaniu innym niż ASCII.
- Opcja `-b` programu `ls`.
- Konwencja zastępowania spacji znakiem podkreślenia.

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ bash param.sh a b c
???
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ bash param.sh a b c
Number of parameters: 3
Parameter 1: 'a'
Parameter 2: 'b'
Parameter 3: 'c'
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ bash param.sh a "" c
???
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ bash param.sh a "" c
Number of parameters: 3
Parameter 1: 'a'
Parameter 2: ''
Parameter 3: 'c'
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ X="" bash param.sh a $X c
???
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ X="" bash param.sh a $X c
Number of parameters: 2
Parameter 1: 'a'
Parameter 2: 'c'
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ X="" bash param.sh a "$X" c
???
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ X="" bash param.sh a "$X" c
Number of parameters: 3
Parameter 1: 'a'
Parameter 2: ''
Parameter 3: 'c'
```

Usuwanie pustych słów

param.sh

```
echo "Number of parameters: $#"
i=1
while (($# > 0))
do
   echo "Parameter $((i++)): '$1'"
   shift
done
```

```
$ X="" bash param.sh a "$X" c
Number of parameters: 3
Parameter 1: 'a'
Parameter 2: ''
Parameter 3: 'c'
```

Puste słowa są usuwane, chyba że są ujęte w cudzysłowy.

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
???
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
The answer is affirmative
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh no
???
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh no
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
???
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
answer.sh: line 1: [: =: unary operator expected
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [ "$1" = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
???
```

Puste argumenty poleceń

answer.sh

```
if [ "$1" = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh "yes and no"
???
```

Puste argumenty poleceń

answer.sh

```
if [ $1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh "yes and no"
test7.sh: line 1: [: too many arguments
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [$1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
???
```

Puste argumenty poleceń

answer.sh

```
if [$1 = yes ]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
test7.sh: line 1: [yes: command not found
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [[$1 = yes ]]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
???
```

Puste argumenty poleceń

answer.sh

```
if [[$1 = yes ]]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh yes
test7.sh: line 1: [[yes: command not found
The answer is not affirmative
```

Puste argumenty poleceń

answer.sh

```
if [[ $1 = yes ]]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
???
```

Puste argumenty poleceń

answer.sh

```
if [[ $1 = yes ]]
then
   echo "The answer is affirmative"
else
   echo "The answer is not affirmative"
fi
```

```
$ bash answer.sh
The answer is not affirmative
```

Wyrażenia logiczne

- `test` i `[` to polecenia wbudowane odpowiadające programom `/usr/bin/test` i `/usr/bin/[`. (Oba programy różnią się nieznacznie.)
- Ostatnim argumentem `[` powinien być `]`.
- Analiza składniowa wyrażenia logicznego jest wykonywana przez powyższe programy, a więc *po* wykonaniu rozwinięć przez basha.
- `[[...]]` jest konstrukcją składniową basha.
- Analiza wyrażenia logicznego wewnątrz `[[...]]` jest wykonywana przez basha *przed* dokonaniem rozwinięć.
- Stare powłoki źle obsługiwały puste słowa.

Typowy idiom w starych skryptach

```
if [ "x$1" = "xyes" ]
then
   ...
```

Błędy w skryptach basha

Rodzaje błędów

- Wywołany program zwrócił niezerowy kod powrotu.
- Instrukcja basha zwróciła niezerowy kod powrotu.
- Skrypt basha zawiera błąd składniowy.

Przykład

```
unset ANSWER
if [ $ANSWER = yes ]
then
   echo Correct
fi
bash: [: =: unary operator expected
```

- Komunikat o błędzie w wyrażeniu pochodzi od polecenia wbudowanego `[`.
- Polecenie `[` (traktowane tak samo jak zewnętrzny program) zwróciło niezerowy kod powrotu.
- Z punktu widzenia basha skrypt jest poprawny.

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
false
echo "Exit code of false: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
???
```

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
false
echo "Exit code of false: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
The next command will fail
Exit code of false: 1
Exit code of true: 0
```

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
nonexistent
echo "Exit code of nonexistent: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
???
```

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
nonexistent
echo "Exit code of nonexistent: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
The next command will fail
fail.sh: line 2: nonexistent: command not found
Exit code of nonexistent: 127
Exit code of true: 0
```

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
fi
echo "Exit code of fi: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
???
```

Błędy a niepowodzenia

fail.sh

```
echo "The next command will fail"
fi
echo "Exit code of fi: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
The next command will fail
fail.sh: line 2: syntax error near unexpected token ‘fi’
fail.sh: line 2: ‘fi’
```

Błędy a niepowodzenia

fail.sh

```
set -o errexit
echo "The next command will fail"
false
echo "Exit code of false: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
???
```

Błędy a niepowodzenia

fail.sh

```
set -o errexit
echo "The next command will fail"
false
echo "Exit code of false: $?"
true
echo "Exit code of true: $?"
```

```
$ bash fail.sh
The next command will fail
```

Selektywna obsługa błędów

`set +o errexit` (default)

```
if wywołanie programu
then
   obsługa błędu, ew. exit nonzero
fi
```

Kodem powrotu instrukcji warunkowej jest kod powrotu ostatniej wykonanej instrukcji w gałęzi `then` bądź `else`.

Idiom (zapożyczony z Perla)

```
wywołanie programu || exit $?
```

Powyższy kod jest jest równoważny, ale bardziej zwięzły niż:

```
set -o errexit
wywołanie programu
set +o errexit
```

Błędy w *globach*

Brak dopasowania w globach

- Domyślnie glob, który nie dopasowuje się do żadnej nazwy pliku rozwija się do siebie samego!
- Uwaga na globy w parametrach programów, np. `find -name *.txt`
- Powyższe polecenie działa dobrze, jeśli w *bieżącym* katalogu *nie ma* plików z rozszerzeniem `.txt`!
- Uwaga na argumenty polecenia `grep`!
- Problemy z testowaniem skryptów.

Opcje globów

- `shopt [-s|-u] failglob` (powinna być domyślna!), `nullglob`, `nocaseglob` i `dotglob`.
- W kontekście `dotglob` nazwy `.` i `..` nie pasują do wzorca `*`.
- Nazwy `.` i `..` zawsze pasują do wzorca `.*`.
- Problem `rm -r .*` i jego rozwiązanie:
  `rm: refusing to remove '.' or '..' directory: skipping '..'`
- Brudne sztuczki: `.??*`. Niestety nie pasuje do `.a`.
  Lepiej `.[^.]*`, choć nie pasuje do `..a`.