

# Systemy operacyjne

## Sprawozdanie - laboratorium 4 „Procesy 1”

Andrzej Kołakowski  
296586

### 1) Implementacja własnej powłoki

#### 1. Przetestowano działanie bazowej wersji powłoki wpisując wybrane polecenia

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ./shellbase
@ ls
@ doprzeniesienia.txt plik1.txt      shell2.c shellbase  sprawozdanie.txt
funcsbases.c          save      shell3    shellbase.c
funcsbases.h          save.c    shell3.c  shell.c
funcs.c               shell     shell4    shellsave
funcs.h               shell2    shell4.c  shellsave.c
echo test

@ test
ps

@  PID TTY          TIME CMD
 2803 pts/0    00:00:00 bash
 8552 pts/0    00:00:00 shellbase
 8553 pts/0    00:00:00 ls <defunct>
 8554 pts/0    00:00:00 echo <defunct>
 8555 pts/0    00:00:00 ps
^C
[root@localhost lab4]#
```

#### Własna powłoka

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ls
doprzeniesienia.txt plik1.txt  shell2.c  shellbase  sprawozdanie.txt
funcsbases.c         save      shell3    shellbase.c
funcsbases.h         save.c    shell3.c  shell.c
funcs.c              shell     shell4    shellsave
funcs.h              shell2    shell4.c  shellsave.c
[root@localhost lab4]# echo test
test
[root@localhost lab4]# ps
  PID TTY          TIME CMD
 2803 pts/0    00:00:00 bash
 8620 pts/0    00:00:00 ps
[root@localhost lab4]#
```

#### Bash

Zaobserwowano, że działanie programu jest bardzo zbliżone do znanego z powłoki systemowej `bash` (z dokładnością do znaku zachęty i ewentualnego kolorowania tekstu).

## 2. Dlaczego znak zachęty nie wyświetla się dopiero po wykonaniu procesu?

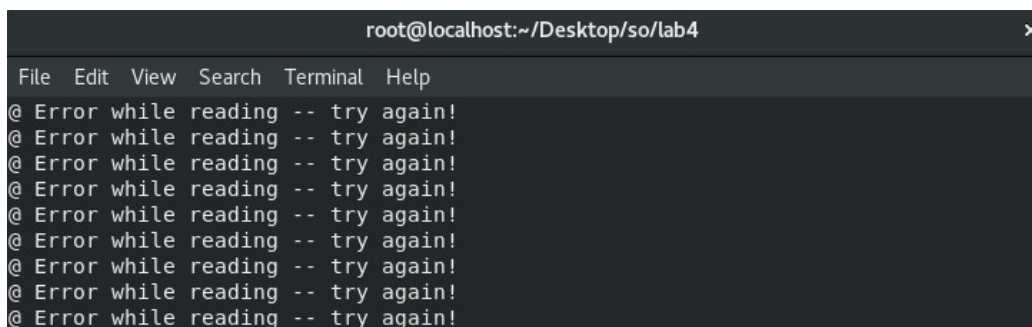
Wykonanie polecenia zadanego powłocie następuje w procesie potomnym, który wykonuje się równocześnie z rodzicem. Rodzic zdąży „zapytać” o kolejne polecenie zanim wykona się proces potomny.

## 3. Rozbudowa istniejących funkcji

- Zmodyfikowano funkcję `executecmds` w taki sposób aby można było zakończyć działanie powłoki poleceniem `exit`.

Zaobserwowano nieprawidłowe działanie!

Próba uruchomienia polecenia które nie istnieje a następnie wyjścia z programu poleceniem `exit` kończy się w następujący sposób:



```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
@ Error while reading -- try again!
```

Użycie polecenia `exit` faktycznie powoduje zakończenie wykonywania procesu, jednak tylko rodzica. Proces potomny wciąż kontynuuje wykonywanie kolejnych linii kodu po powrocie z nieudanego wywołania `execvp(3)`. Konieczny zatem jest również kod kończący jego działanie pod wywołaniem `execvp`.

- Zmodyfikowano funkcję `executecmds` w taki sposób, aby oczekiwała na zakończenie się uruchomionego procesu.

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ./shellbase3

@ ls
doprzeniesienia.txt  plik1.txt  shell2.c  shellbase  shellbase.c
funcsbase.c          save       shell3    shellbase2  shell.c
funcsbase.h          save.c     shell3.c  shellbase2.c  shellsave
funcs.c              shell      shell4    shellbase3  shellsave.c
funcs.h              shell2     shell4.c  shellbase3.c  sprawozdanie.txt

@ echo test
test

@ ps
  PID TTY          TIME CMD
 9522 pts/1        00:00:00 bash
 9848 pts/1        00:00:00 shellbase3
 9851 pts/1        00:00:00 ps

@ ^C
[root@localhost lab4]#
```

Warto zauważyć, że znak zachęty pojawia się dopiero po zakończeniu procesu.

- Zmodyfikowano funkcję `executecmds` w taki sposób aby do zmiennej `int proces` zapisywała wartość oznaczającą sposób zakończenia się ostatniego procesu (1 w przypadku pomyślnego zakończenia, 0 w przeciwnym wypadku) a następnie wyświetlała kod wyjścia procesu.

Otrzymany efekt:

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ./shell5

@ echo test
test
Child terminated normally with status: 0

@ ls
doprzeniesienia.txt  save       shell3.c  shellbase2  shellsave
funcsbase.c          save.c     shell4    shellbase2.c  shellsave.c
funcsbase.h          shell      shell4.c  shellbase3  sprawozdanie.txt
funcs.c              shell2     shell5    shellbase3.c
funcs.h              shell2.c  shell5.c  shellbase.c
plik1.txt            shell3     shellbase  shell.c
Child terminated normally with status: 0

@ ls hhh
ls: cannot access 'hhh': No such file or directory
Child terminated normally with status: 2

@ cat aaa
cat: aaa: No such file or directory
Child terminated normally with status: 1

@ █
```

Porównanie z bashem:

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# echo test
test
[root@localhost lab4]# echo $?
0
[root@localhost lab4]# ls hhh
ls: cannot access 'hhh': No such file or directory
[root@localhost lab4]# echo $?
2
[root@localhost lab4]# cat aaa
cat: aaa: No such file or directory
[root@localhost lab4]# echo $?
1
[root@localhost lab4]#
```

- Zmodyfikowano funkcję `parsecmd`, w taki sposób, aby poprawnie interpretowała operatory `||` i `&&`, oraz funkcję `executecmds`, w taki sposób, aby uruchamiała procesy zgodnie z podanymi operatorami `&&` oraz `||`.

Przykładowe wyjście wraz z dodatkowymi informacjami diagnostycznymi:

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
@ ls nieistnieje && echo OK || ps
Parsed command(s):
Command 1:
argv[0]: ls
argv[1]: nieistnieje
argv[2]: (null)
Command 2:
argv[0]: echo
argv[1]: OK
argv[2]: (null)
Command 3:
argv[0]: ps
argv[1]: (null)

Command=ls, Conjunction=0
ls: cannot access 'nieistnieje': No such file or directory
Child terminated normally with status: 2

Command=ps, Conjunction=1
  PID TTY          TIME CMD
  9522 pts/1        00:00:00 bash
 10656 pts/1        00:00:00 shelldebug
 10660 pts/1        00:00:00 ps
Child terminated normally with status: 0
```

#### 4. Programy końcowe

*shell.c, shelldebug.c*

## 2) Ustawianie limitów procesu

1. Rozbudowa programu – dodanie obowiązkowego argumentu wywołania programu (filename)

2. Rozbudowa programu – dodanie opcjonalnego argumentu wywołania programu (bytes)

Docelowy sposób wywołania programu: `./save [bytes] filename`

3. Test działania programu

a) Zapis 100 bajtów do pliku o nazwie tmp1.txt

```
root@localhost:~/Desktop/so/lab4 x
File Edit View Search Terminal Help
[root@localhost lab4]# ./save tmp1.txt
RLIMIT_FSIZE: cur=-1, max=-1
Writing 100 bytes into tmp1.txt file...
[root@localhost lab4]# cat tmp1.txt | wc -c
100
[root@localhost lab4]#
```

b) Zapis 53 bajtów do pliku o nazwie tmp2.txt

```
root@localhost:~/Desktop/so/lab4 x
File Edit View Search Terminal Help
[root@localhost lab4]# ./save 53 tmp2.txt
RLIMIT_FSIZE: cur=-1, max=-1
Writing 53 bytes into tmp2.txt file...
[root@localhost lab4]# cat tmp2.txt | wc -c
53
[root@localhost lab4]#
```

4. Rozbudowa programu – sprawdzenie czy podano odpowiednią ilość argumentów

```
root@localhost:~/Desktop/so/lab4 x
File Edit View Search Terminal Help
[root@localhost lab4]# ./save
Bad number of arguments, usage: ./save [bytes] file
[root@localhost lab4]# echo $?
1
[root@localhost lab4]# ./save 53 tmp2.txt tmp3.txt
Bad number of arguments, usage: ./save [bytes] file
[root@localhost lab4]# echo $?
1
[root@localhost lab4]#
```

## 5. Dodatkowe testy

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ./save tmp1.txt || echo FAIL
RLIMIT_FSIZE: cur=-1, max=-1
Writing 100 bytes into tmp1.txt file...
[root@localhost lab4]# ./save 200 tmp1.txt && echo OK
RLIMIT_FSIZE: cur=-1, max=-1
Writing 200 bytes into tmp1.txt file...
OK
[root@localhost lab4]# ./save || echo FAIL
Bad number of arguments, usage: ./save [bytes] file
FAIL
[root@localhost lab4]# ./save && echo OK
Bad number of arguments, usage: ./save [bytes] file
[root@localhost lab4]#
```

## 6. Modyfikacja „własnej powłoki” – dodanie miękkiego limitu maksymalnej wielkości tworzonych plików (50 bajtów)

## 7. Sprawdzenie, czy ustawiony limit jest dziedziczony przez procesy potomne

Do testów wykorzystano własną powłokę z dodanym miękkim limitem wielkości plików.

```
root@localhost:~/Desktop/so/lab4
File Edit View Search Terminal Help
[root@localhost lab4]# ./shellsave
NEW RLIMIT SET

@ ./save 10 tmp3.txt
RLIMIT_FSIZE: cur=50, max=-1
Writing 10 bytes into tmp3.txt file...
Child terminated normally with status: 0

@ wc -c tmp3.txt
10 tmp3.txt
Child terminated normally with status: 0

@ ./save tmp4.txt && ./save tmp5.txt
RLIMIT_FSIZE: cur=50, max=-1
Writing 100 bytes into tmp4.txt file...
Child terminated by signal

@
```

Zaobserwowano efekt inny niż przedstawiony w instrukcji do laboratorium.

W instrukcji zasugerowano, że plik `tmp4.txt` zostanie utworzony, jednak rozmiar zostanie obcięty z domyślnego 100 bajtów do 50.

Patrząc jednak do manuala `getrlimit(2)` dowiadujemy się, że:

*Attempts to extend a file beyond `[RLIMIT_FSIZE]` result in delivery of a `SIGXFSZ` signal. By default, this signal terminates a process, but a process can catch this signal instead, in which case the relevant system call (e.g., `write(2)`, `truncate(2)`) fails with the error `EFBIG`.*

Co jest zgodne z tym, co zaobserwowano.

Niemniej jednak, przeprowadzony test pokazuje, że proces potomny (`save`) odziedziczył limit po powłoce.

## 8. Programy końcowe

*save.c, shellsave.c*